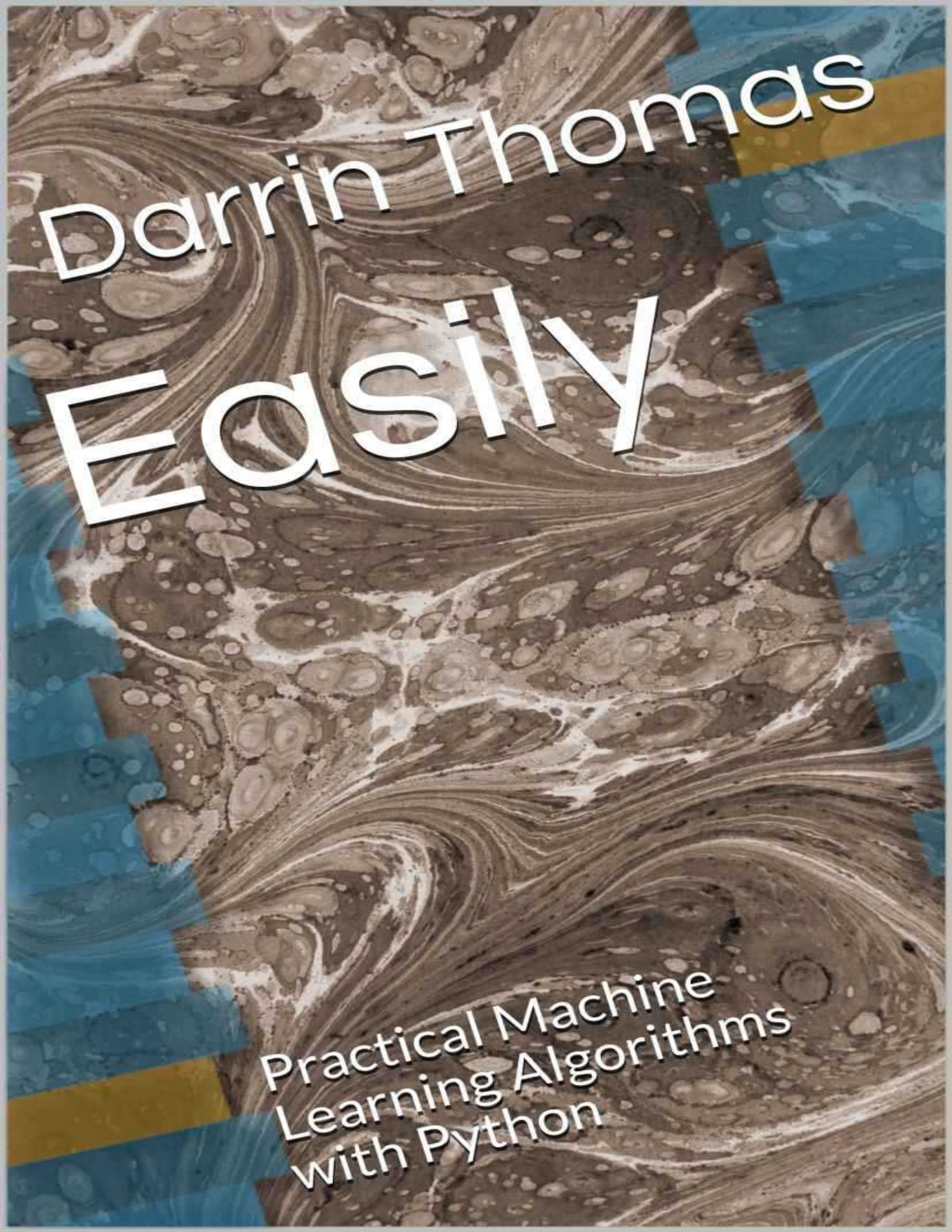

Darrin Thomas
Easily
Practical Machine Learning Algorithms with Python

Easily

Practical Machine Learning

Algorithms with Python

Darrin Thomas

Saraburi, Thailand

Layout: Darrin Thomas

Photo Researcher: Darrin Thomas

Cover Design: Darrin Thomas

Copyright 2019 ERT Group, publishing as SuJinSoLa, Saraburi, Thailand.
All rights reserved.

ISBN-FIX ME

About the Author

Darrin Thomas, PhD, grew up in Sacramento, California and has over ten years of experience as a teacher and lecturer from Kindergarten to graduate school. He completed his bachelor and master degree in saxophone performance at California State University Sacramento. After working as a substitute teacher, he completed a credential in teaching at Pacific Union College. He then worked as a music teacher before moving to Thailand to work as a lecturer in Education/Psychology Department at Asia-Pacific International University (APIU).

While overseas, Dr. Darrin completed his master degree in education at APIU. He then moved to the Philippines and completed his doctoral degree in education at Adventist International Institute of Advanced Studies. Currently, Dr. Darrin is a lecturer at Asia-Pacific International University. His enthusiasm for machine learning has led to works involv-ing many different algorithms applied in an educational context. His blog is at Educational Research Techniques

Contents

Table of Contents

Chapter 1 Decision Trees-Classification

Chapter 2 Decision Trees-Numeric Prediction

Chapter 3 Random Forest-Classification

Chapter 4 Random Forest-Numeric Prediction

Chapter 5 K Nearest Neighbor-Classification & Numeric Prediction

Chapter 6 Support Vector Machines-Classification

Chapter 7 Support Vector Machines-Regression

Chapter 8 Artificial Neural Networks

Chapter 9 K-Means

Chapter 10 Assessing Classification Model Performance

Chapter 11 Assessing Regression Model Performance

Chapter 12 Model Improvement Strategies

Chapter 13 Combining Algorithms

Preface

Using Python the programming language in the discipline of data science is standard practice in industry and even at times in academia. For this reason, this book was written to provide a simple, easy to understand, condense resource that deals with a variety of commonly used machine learning algorithms along with ways to judge and improve models. The goal was always to show what to do rather than talk a lot about how to do it.

One assumption of this text is that the readers is already familiar with Python and that they are able to comprehend and interpret the code in the examples. Of course, as the author of this text, I make no claim to being the ultimate authority on the use of Python and the details of using this language. However, having said this, I do know how to get things done when using this language and want to share what I know to help others.

In addition, it also assumed that the reader is already familiar with the realm of data science and machine learning. As such, there will not be a detailed description of the data science pipeline, classification, regression, data mugging, etc. This book is focused specifically on using the algorithms and not teaching the basics of data science.

Therefore, the target audience for this book are individuals with some background in both coding, statistics, and basic concepts of data science who are looking to extend their skill into using Python for statistical analysis to deal with problems commonly associated with data science.

Throughout this text the anaconda installation of Python is used. If you use this distribution of Python, there may be times when you need to install additional libraries that are used in the demonstration code of this book. These additional libraries can be downloaded through open your terminal in your computer and using the pip install followed by the name of the library.

Chapters 1-8 deal with supervised learning for classification and regression using decision trees, random forest, support vector machines, K Nearest Neighbors, and artificial neural networks. Chapter 9 provides an example of unsupervised learning using K-Means. Chapters 10-12 deal with assessing

classification and regression models and how to improve models based on the assessment of them. Lastly, chapter 13 provides an example of the integration of all the ideas of this book through combining algorithms to make classification predictions.

Chapter 1: Decision Trees-Classification

In this chapter, you will learn the following...

• What a decision tree is.

• How to develop a decision tree for classification

• How to interpret decision tree results

Introduction

Decision trees are used for classification or numeric prediction. The goal of the analysis when employing decision trees is to sub-divide the sample into subsets that have high within group similarity.

A decision tree consists of a diagram called a tree structure. A tree structure has root node, decision nodes, branches, and leaf nodes. At the top is the root node, which is also the initial decision the decision tree makes. Following the root node, you will often have additional splits with decision nodes leading finally to terminal or leaf nodes. Figure 1.1 provides a visual of a decision trees.

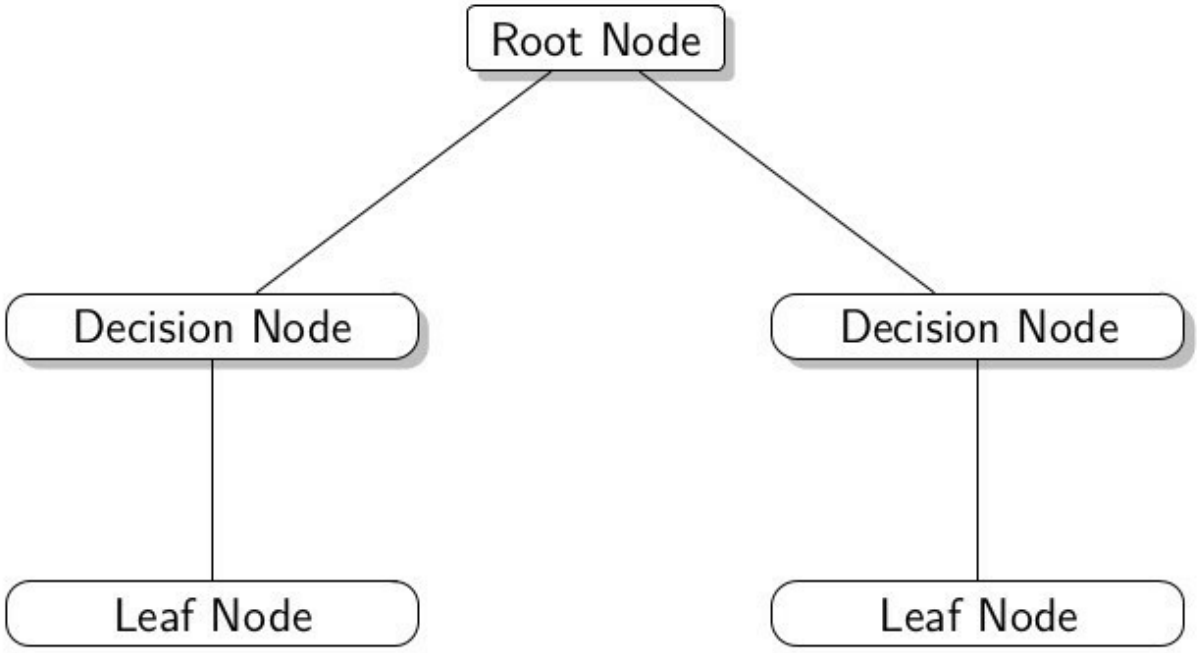


Figure 1.1: Decision Tree Structure

The output of a decision tree can also be in text. However, this is harder to interpret and understand so we will make a visual of a decision tree when we get to the analysis part.

Decision Trees Details

There are many different algorithms that can be used to make a decision trees. Examples include C5.0, CTree, and Quest. Different algorithms can provide different performance based on how the decision tree is selected. However, this is a bit of a more advance concept. For our purposes we will use the default algorithm that Python employees.

For classification, the main way that the sample is divided is through a metric called entropy, which ranges from 0 to 1. The lower the entropy the more homogeneous a potential sub-division is. When the purpose is numeric prediction the metric can be the rmse (root mean squared error)or r^2.

One of the primary benefits of decision trees is their highly flexible nature. There are no issues with missing data or non-normality. Furthermore, decision trees do not have to use all of the variables when making the tree. This is different from other algorithms that must include all variables. Lastly, it is not necessary to have a strong background in math to interpret decision trees.

Despite these advantages there are some difficulties with decision trees. It is easy for a decision tree to become highly complex. This is an indication of overfitting which means that the model is custom fitting to the sample and cannot be generalize to new data. Highly complex models are difficult to explain and understand. There are ways to address this problem but they will be discussed later.

Our goal for this chapter is to make a classification decision tree that is going to predict the health status of people with cancer. In the next section, we will deal with the preparation of the data.

Data Preparation

Below are the libraries we will use for the rest of this chapter. As we use them they will be explained.

```
import pandas as pd
from pydataset import data
from sklearn.model_selection import train_test_split
from sklearn import metrics
from sklearn import tree
```

We are going to use the cancer dataset from the pydataset library. In the code below, we load the cancer dataset into an object called df. We then use the `.info()` method to learn more about the variables that are available. Lastly, we drop any NAs by using the `.dropna()` method. Below is the code and output.

```
In [4]: df = data('cancer')
   ...: df.info()
   ...: df=df.dropna()
<class 'pandas.core.frame.DataFrame'>
Int64Index: 228 entries, 1 to 228
Data columns (total 10 columns):
inst          227 non-null float64
time          228 non-null int64
status        228 non-null int64
age           228 non-null int64
sex           228 non-null int64
ph.ecog       227 non-null float64
ph.karno      227 non-null float64
pat.karno     225 non-null float64
meal.cal      181 non-null float64
wt.loss       214 non-null float64
dtypes: float64(6), int64(4)
memory usage: 19.6 KB
```

You can see that there are ten variables in this dataset with a total of 228 examples. All of the data is coded as continuous data. This may appear to be a problem but the status variable is coded 1 and 2 with one meaning alive for our purposes and 2 meaning dead. Below is a breakdown of the categories in the `status` variable.

```
In [15]: pd.crosstab(index=df["status"],columns="count")
Out[15]:
col_0    count
status
1           47
2          120
```

Remember the originally data set had 228 rows. However, we now only have 167 rows of data. This is because we removed the rows that had NAs. There are other ways to handle missing data but for our purposes this is the simplest way for now.

Below are the correlations between the continuous variables.

```
In [41]: df[:-3].corr()
Out[41]:
                inst      time       age       sex   ph.ecog  ph.karno  pat.karno  meal.cal   wt.loss
inst        1.000000  0.044798  0.033832  0.053096  0.037900  0.008529   0.052077  0.096285 -0.170606
time        0.044798  1.000000 -0.071795  0.125654 -0.184342  0.087816   0.181841  0.079204  0.026106
age         0.033832 -0.071795  1.000000 -0.132516  0.300907 -0.315751  -0.239333 -0.243189  0.045552
sex         0.053096  0.125654 -0.132516  1.000000 -0.016785 -0.002290   0.078451 -0.175802 -0.167133
ph.ecog     0.037900 -0.184342  0.300907 -0.016785  1.000000 -0.821471  -0.550445 -0.108670  0.176534
ph.karno    0.008529  0.087816 -0.315751 -0.002290 -0.821471  1.000000   0.533925  0.055906 -0.128983
pat.karno   0.052077  0.181841 -0.239333  0.078451 -0.550445  0.533925   1.000000  0.172870 -0.177108
meal.cal    0.096285  0.079204 -0.243189 -0.175802 -0.108670  0.055906   0.172870  1.000000 -0.108475
wt.loss    -0.170606  0.026106  0.045552 -0.167133  0.176534 -0.128983  -0.177108 -0.108475  1.000000
```

The -3 in brackets removed the categorical variable status. It appears that the correlations between most of the variables are weak. In terms of exploratory analysis this is all we will do. Decision trees are robust against non-normality so we are not to worried about exploring the characteristics of the data.

Below is the code for creating the independent and dependent variables.

```
X=df[['time','age',"sex","ph.ecog",'ph.karno','pat.karno','meal.cal','wt.loss']]
df['status']=df.status.replace(1,'censored')
df['status']=df.status.replace(2,'dead')
y=df['status']
```

Here is what is happening

- In the first line we are placing all of our independent variables into an object called X.

- In the second two lines, we are replacing the numerical values in the status variable with text. This is simply to aid the interpretation later.

- In the last line, we are setting the status as the dependent y variable.

The final step in the data preparation is to create our training and test sets. In data science, it is common to divide your dataset into at least 2 sets. One set is called the training set and the other is called the test set. Sometimes there is a third set called the validation set but that is beyond the scoop of this text.

The training set is used to set the parameters of your model and the test set is used to see if your model will generalize to more data. Generally, you want the performance of your training set to be a little better than your test. This is an indication that your model will generalize. Being able to generalize is important because we want our data to have similar performance when given new data.

It is common to divide the dataset 70/30 with 70% of the data being used for training and 30% of the data used for testing. However, there are other ratios that are used and it can become quite extreme if you have a large amount of data.

To complete this process we will need a train and test for the X variables and a train and test for the y variable meaning we will need 4 datasets altogether. We will use the train_test_split() method from sklearn.

Below is the code for this

```
x_train, x_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
                                                    random_state=0)
```

The test_size argument indicates what proportion of the data should be set aside for testing. The random_state argument is for setting the seed for being able to reproduce the results.

We can now move to model development.

Model Development

First, we will create an instance of our decision tree classifier and then we will create our actual model using the `.fit()` method with the train data. This function fits the model to the data. Below is the code.

```
clf=tree.DecisionTreeClassifier(max_depth=3)

clf=clf.fit(x_train,y_train)
```

The `max_depth` argument indicates how deep the decision tree will go. For purposes of demonstration we will only go to a depth of three. This means there will be three levels to the decision tree. This is one example of preventing overfitting and encouraging simple models that generalize.

Below we make a visual of the decision tree. There are some additional libraries that are needed for this and they are found in the code below followed by the actual decision tree.

```
In [62]: from sklearn.externals.six import StringIO
    ...: from IPython.display import Image
    ...: from sklearn.tree import export_graphviz
    ...: import pydotplus
    ...: dot_data = StringIO()
    ...: export_graphviz(clf, out_file=dot_data,
    ...:                 filled=False,
rounded=True,feature_names=list(x_train.columns.values),
    ...:                 class_names=list(y_train),special_characters=True)
    ...: graph = pydotplus.graph_from_dot_data(dot_data.getvalue())
    ...: Image(graph.create_png())
```

Each square in the decision tree contains the following information (with exception). This information is explained going from top to bottom.

1. The variable on which the decision is based.

2. The level of purity. The closer to 0 the better.

3. The number of samples in the node.

4. The classification of the node samples.

5. The majority class in the node.

As an example, the highest node uses the variable time being less than or greater than 171. The gini is 0.4, the samples are 116 (the sample size of the train dataset), the values are 32 are alive while 84 are dead, and the classification is "dead" based on the majority of the values in this node.

The highest level is called the root node. The next two levels are decision nodes and the last level is the leaf nodes. If the criteria in the node is true you go down and to the right. If the criteria is false you go down and to the left. Starting from the top, below is one way to read the tree using only criteria that are true.

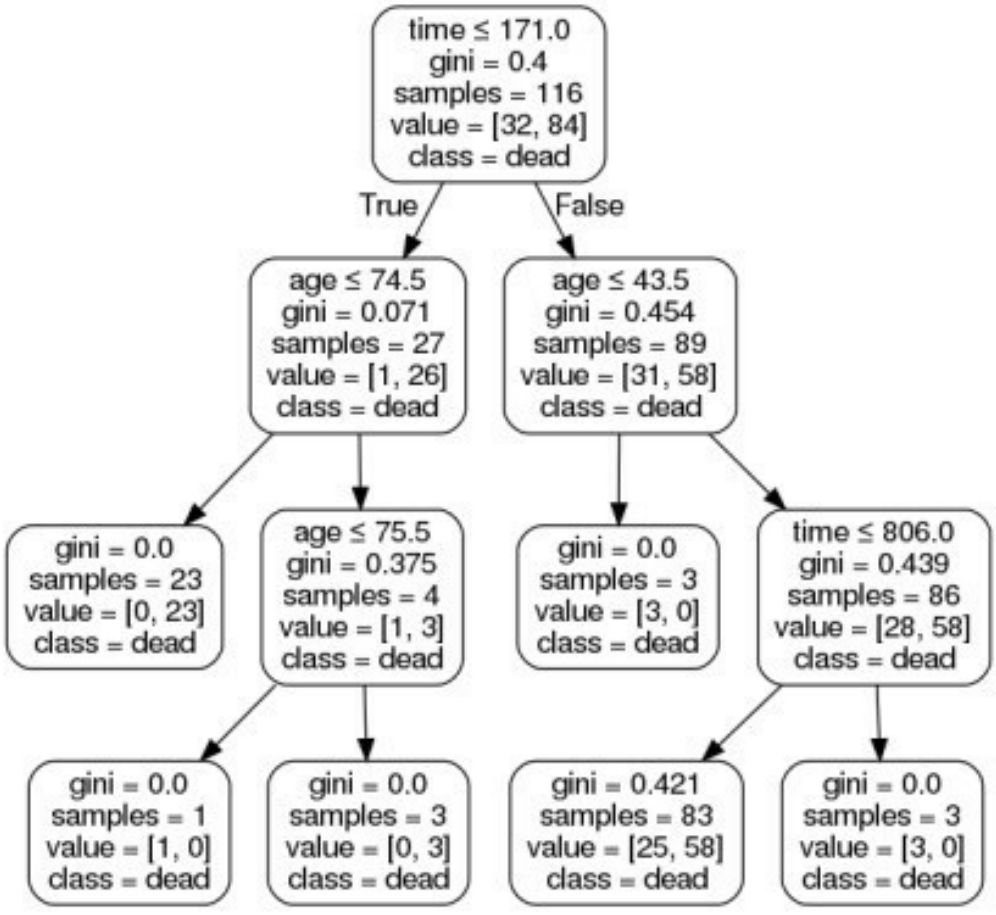


Figure 1.2: Decision Tree

1. If someone has spent 171 minutes or less AND

2. They are 74.5 years or less AND

3. Then they are dead

There are other ways to read this that you can explore on your own. Another source of information that can be useful to know is the variable importance. This metric tells you which variables were most useful in developing the decision tree. Below is the code and output for how to do this.

```
In [7]: import numpy as np
   ...: results=dict(zip(x_train.columns,
   ...:                  np.round(clf.feature_importances_,2)))
   ...: sorted(results.items(),
   ...:        key=lambda x: x[1],reverse=True)
Out[7]:
[('time', 0.96),
 ('age', 0.04),
 ('sex', 0.0),
 ('ph.ecog', 0.0),
 ('ph.karno', 0.0),
 ('pat.karno', 0.0),
 ('meal.cal', 0.0),
 ('wt.loss', 0.0)]
```

This code is a little complex compared to what we have done before, I will attempt to explain it below.

• The first line loads the library called numpy and call it np.

• Next, we create a dictionary. Inside the dictionary we take the columns from our `x_train dataset` and taken the rounded (2 decimal places) of the `feature_importances` function. All this is saved in an object called `results`.

• In the fourth and fifth line we use the sorted method to sort the information in the `results` object by the feature importance.

The output indicates that time is the most important feature or variable followed by age. This makes sense because if you look at figure 1.2 you can see that only time and age are used to split the data. One of the main benefits

of the variable importance metric is it gives you some sense of the relative value of each variable in classifying samples.

Next, we need to see how our model is performing by making predictions and looking at the accuracy. In the code below, we use the `.predict()` method to make classification predictions using are clf model with the x_train dataset. This action is stored in an object called `y_pred_train`.

```
y_pred_train=clf.predict(x_train)
```

We can now assess the strength of our model using accuracy as the criteria. We will make a matrix that will tell us how many times our model predicted correctly and incorrectly. This is done using the `.crosstab()` method. In addition, we will also calculate the accuracy using the `metrics.accuracy_score()` method (the round function is just for rounding the accuracy to two dig-its). The code is below followed by the output.

```
In [66]: print (pd.crosstab(y_train,y_pred_train))
    ...: round(metrics.accuracy_score(y_train,y_pred_train),2)
    ...:
col_0     censored  dead
status
censored         7    25
dead             0    84
Out[66]: 0.78
```

The diagonal from the top left to the bottom right are examples that were correctly classified. Seven people with a status of censored (alive) were classified correctly while 84 people with a status of dead were classified correctly. 25 people who were censored were classified as dead and 0 dead were classified as censored. Overall, the accuracy was 78%. Additional metrics will be discussed in a later chapter along with more information on the matrix we made. We are able to determine this information because we know the actual y values of the training set. This information is called the ground truth. What we created were the prediction that came from our model. The ground truth is being compared to our predictions to give us the accuracy of our model. Now, we will now see how are model does with the test data.

Model Testing

We have already seen all the code that we need to test the model. The only difference is instead of using the train dataset we will use the test dataset. Our model is not familiar with this data so it provides us with an insight into how our model will generalize with new data. Below is the code and the output.

```
In [69]: y_pred=clf.predict(x_test)
    ...: print (pd.crosstab(y_test,y_pred))
    ...: round(metrics.accuracy_score(y_test,y_pred),2)
col_0     censored  dead
status
censored         2    13
dead             1    35
Out[69]: 0.73
```

In the first line we are using the `.predict()` method again but with the `x_test dataset`. The use of the `x_test dataset` is repeated for the next two lines of code. Next, is the matrix which gives an insight into the correct and incorrect predictions by comparing the y values of the test set with the predicted values. The overall accuracy is 73% which is lower than our training data. This is normal as our model has not seen this data, which usually lowers the performance.

Conclusion

Decision trees are a great tool for making easy to understand models that are fairly good at making predictions. The primary goal is to make sub-divisions in the data that are as pure as possible based on the variables that are available in the dataset. There are some concerns with this type model yet it is one of money tools available to the data scientist.

Chapter 2: Decision Trees-Numeric Prediction

In this chapter, you will learn the following...

- Explain the characteristics of regression trees
- Explain the pros and cons of regression trees
- How to develop a decision tree for numeric prediction
- How to interpret decision tree for numeric prediction

Introduction

Regression trees are decision trees that have a continuous variable as the dependent variable. Despite their name, regression trees do not involve any regression. Instead, the mean is predicted at each leaf in the decision tree. For our purposes, we will use the term numeric prediction instead of regression tree in order to avoid confusion over the lack of the use of regression.

The decision on when to split the tree is based on the use of a technique called recursive partitioning. The split is made based on what reduces the error the most. The end goal is always to produce nodes that are as homogeneous as possible.

Pros & Cons of Numeric Prediction Trees

Some of the strengths of numeric decision trees are as follows

- Straightforward to interpret
- Can handle non-normal and non-linear data
- Feature selection is done automatically
- No prior modeling necessary

Some of the weaknesses include the following

- The use of the mean for making leaves affects the accuracy of the predictions because the mean is not always the best fit for each split.
- Recursive partitioning is a greedy algorithm. This means that when splitting the data it does not consider the consequences, which leads to a lot of superficial splits.
- Requires more data for the training set in comparison to classification.

We can now move to the data preparation for our model. We will use the same dataset as the last chapter. However, our goal this time will be to predict a person's age based on the other variables in the dataset `cancer`.

Data Preparation

The data preparation is mostly the same. There are some different libraries that we will use when compared to the previous chapter. Below are the initial libraries that we will use.

```
import pandas as pd
import numpy as np
import seaborn as sns
from pydataset import data
from sklearn.model_selection import train_test_split
from sklearn import tree
import matplotlib.pyplot as plt
from sklearn.metrics import mean_squared_error
```

The correlations of the variables are in the previous chapter. Therefore, we will now make our dataset for the independent variables and our dataset for the dependent variable. The code is below.

```
df = data('cancer')
df=df.dropna()
X=df[['time',"sex","ph.ecog",'ph.karno','pat.karno',
      'meal.cal','wt.loss']]
y=df['age']
```

You can see that the main difference is that age is now the dependent variable. Next, we will move to preparing our train and test sets. The code is below and is the same as in the previous chapter.

```
x_train, x_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
                                                    random_state=0)
```

We can now move to model development.

Model Development

There is little to explain since most of the code is the same as last chapter. Below we create our actual decision tree model.

```
reg=tree.DecisionTreeRegressor(max_depth=3)
reg=reg.fit(x_train,y_train)
```

Next, we make a visual of the results in figure 2.1.

```
from sklearn.externals.six import StringIO
from IPython.display import Image
from sklearn.tree import export_graphviz
import pydotplus
dot_data = StringIO()
export_graphviz(reg, out_file=dot_data,
                filled=False, rounded=True,
                feature_names=list(x_train.columns.values),
                class_names=list(y_train),special_characters=True)
graph = pydotplus.graph_from_dot_data(dot_data.getvalue())
Image(graph.create_png())
```

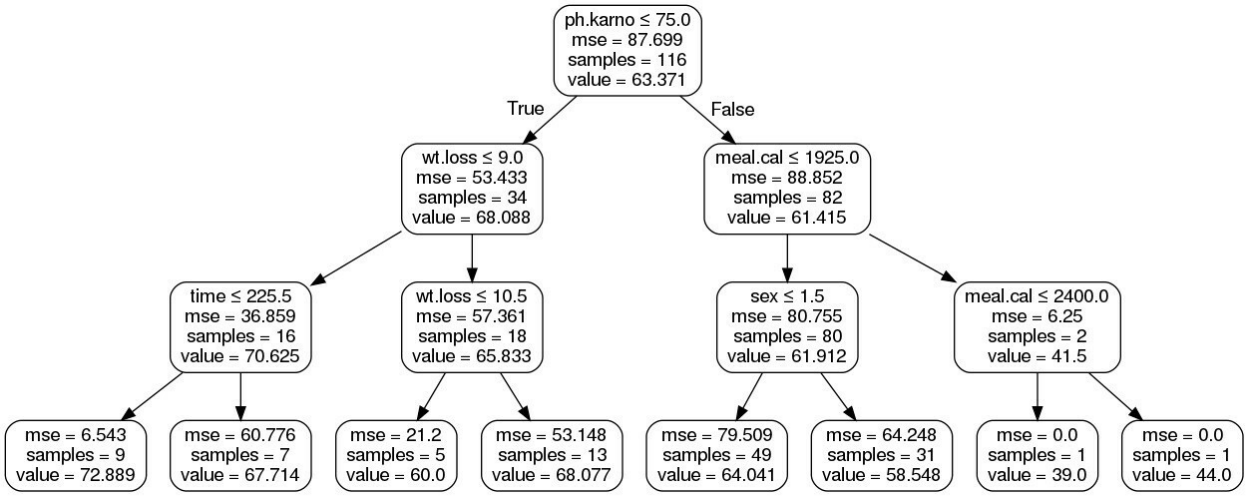


Figure 2.1: Regression Decision Tree

There are some subtle differences in this decision tree compared to the classification decision tree. Each node's decision node begins with the variable. However, the next value is now the MSE or mean squared error

(metric for error) rather than the gini coefficient. Next, we have the number of samples that meet the criteria followed by the mean of our dependent variable in the line labeled value.

Naturally, it would not be practical to explain every node in the decision tree. However, here is an explanation of the far left nodes below.

1. If the ph.karno are less then 75 AND
2. The weight lost is less then 9 AND
3. The time is less than 225 AND
4. THEN the person is about 73 years old

We will now try to assess the importance of each variable. The code and output are as follows.

```
In [17]: results=dict(zip(x_train.columns,
    ...:                     np.round(reg.feature_importances_,2)))
    ...: sorted(results.items(),
    ...:        key=lambda x: x[1],reverse=True)
Out[17]:
[('ph.karno', 0.36),
 ('meal.cal', 0.27),
 ('sex', 0.19),
 ('wt.loss', 0.14),
 ('time', 0.04),
 ('ph.ecog', 0.0),
 ('pat.karno', 0.0)]
```

You can see that ph.karno was the most important followed by meal.cal.

We can now assess the predictive power of our model with the training

data. Since this is numeric prediction the process is slightly different. To make things as clear as possible we are going to do the following four

things.

1. Create the prediction model
2. Calculate the correlation between the actual and predicted values
3. Make a scatterplot of the actual and predict values
4. Calculate the mean squared error of the model

First we create the prediction model. The code is below

```
y_pred_train=reg.predict(x_train)
```

Next, we will calculate the correlation between the predicted and actual values.

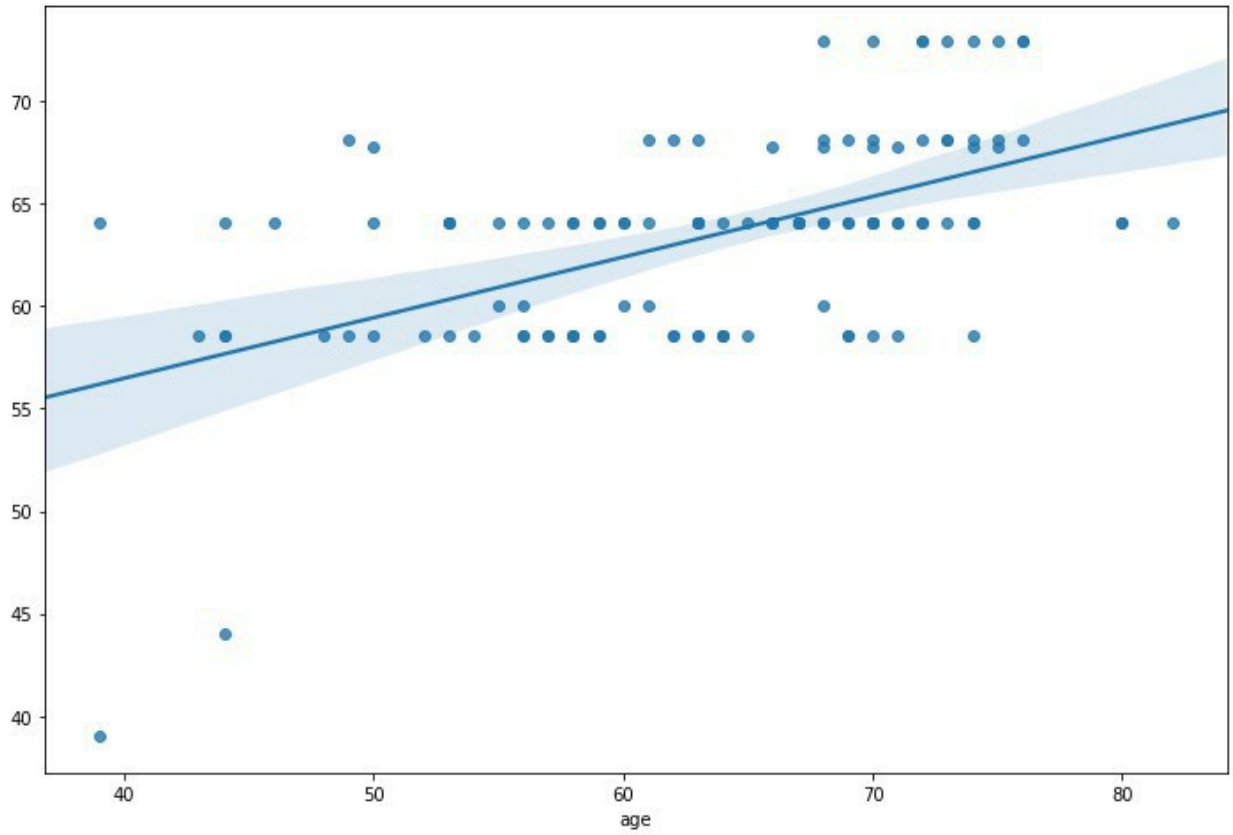
```
np.corrcoef(y_pred_train,y_train)
```

```
0.54
```

The correlation appears moderate. We will confirm this linear relationship with the scatterplot below.

```
fig, ax = plt.subplots(figsize=(12,8))
```

```
sns.regplot(y_train, y_pred_train)
```



Well this does not look like it is linear. However, one advantages of decision trees is they can fit non-linear data. Therefore, even though this looks bad it may not impact the performance of the model. Finally, we will calculate the mean squared error (MSE).

```
mean_squared_error(y_train, y_pred_train) Out[24]: 61.8
```

The MSE is a relative metric. In other words, it only has meaning when this value is compared to another model's MSE. The lower the MSE the better is the general rule. We can now move to assess our model with the test data.

Model Testing

The steps for model testing are exactly the same as for model development. We will first predict with the test data, calculate the correlation, make the scatterplot, and calculate the MSE. In the code below, we predict and calculate the correlation.

```
y_pred=reg.predict(x_test)
```

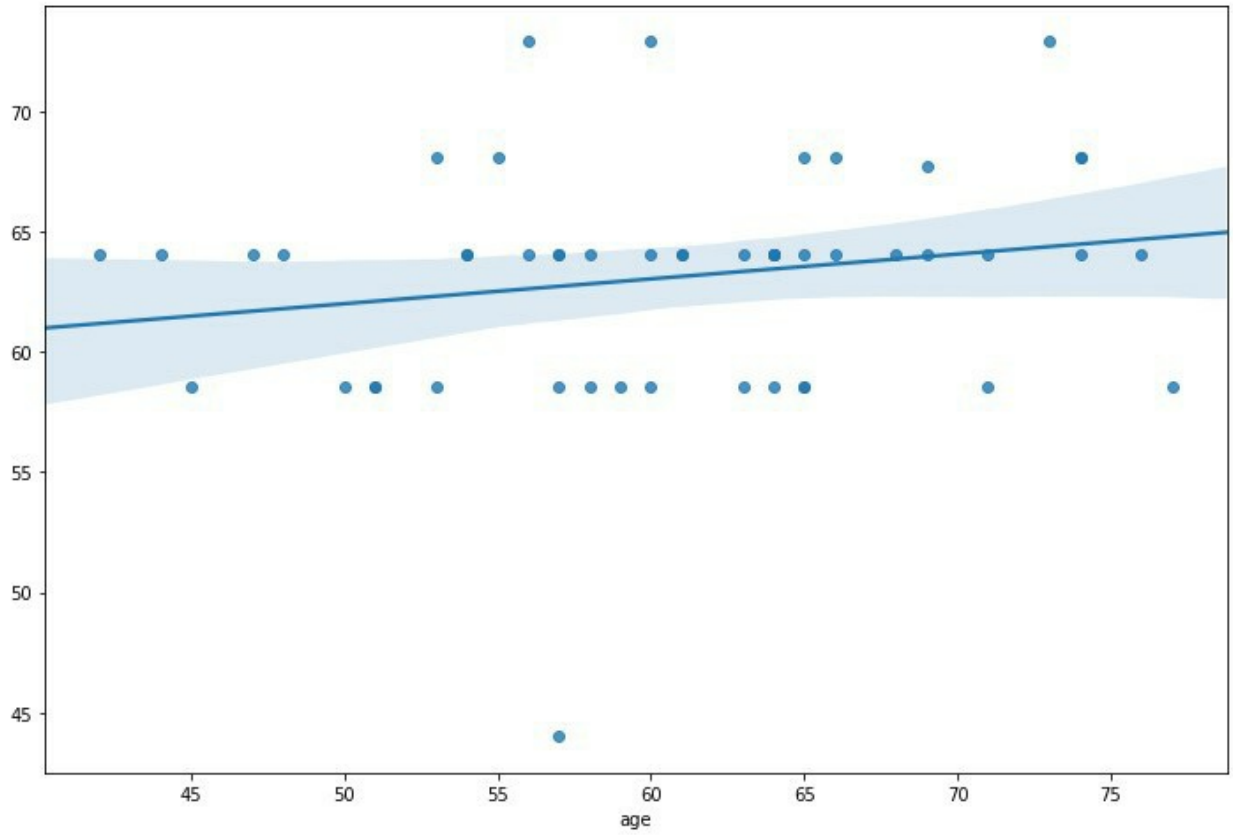
```
np.corrcoef(y_pred,y_test)
```

```
0.18
```

The correlation is much lower. This means that something may not be right. Next, we create the scatter plot.

```
fig, ax = plt.subplots(figsize=(12,8))
```

```
sns.regplot(y_test,y_pred)
```



The scatterplot confirms the weak relationship. We will now calculate the

MSE.

```
mean_squared_error(y_pred,y_test)
```

```
Out[31]: 85.24
```

The MSE is worst compared to the train data. However, this is expected.

Conclusion

In this chapter, we discussed and demonstrated how to create and assess a numeric prediction decision tree model. Being a decision tree, numeric prediction trees have all the strengths and weaknesses of a classification decision tree. Therefore, it is important to consider when ti would beneficial to use this algorithm.

Chapter 3: Random Forest-Classification

In this chapter, you will learn the following...

- Explain the characteristics of random forest.
- Explain the pros and cons of random forest.
- How to use random forest for classification.
- How to interpret the results of random forest classification.

Introduction

Decisions trees provide excellent visualizations and can be easy to understand at times. However, a glowing concern with decisions trees is that it is easy for them to overfit the data. By overfitting we mean that the model follows the training data so well that it does not fit any other dataset. An analogy would be the difference between having clothes custom tailor for you or buying "one-sizes-fits-all." The tailored clothing would overfit you, which means almost no one else can wear it, while the one-size-fits-all will fit everyone somewhat. When we are making models we want to find a middle ground between the tailored clothing and the one-size-fits-all approach.

One solution to the weakness of overfitting is random forest. Random forest involves making not one decision tree but rather hundreds, thousands, or more trees. The results of all the different trees or forest are combined to provide an aggregated model. The criteria for how the trees are made can vary a great deal but some approaches that are used include the following.

- Different trees use different subsets of the sample
- Different trees use different variables from the dataset
- Different number of trees developed
- etc.

This variation in the tree developed helps to eliminate a large amount of the overfitting that can happen with an individual model. By now you may understand why it is called random forest. This is due to the way that a large number of trees (the forest) are made arbitrarily (random).

For classification, the predictions are determined by simple majority vote. For example, if I make three trees and there are two votes "yes" and one vote "no" the aggregated model will predict "yes". If the random forest is regression, the mean is calculated from all the trees that are created.

Pros & Cons of Random Forest

The benefit of random forest have already been discussed with one of the main benefits being the reduction in the risk of overfitting through developing so many trees. In addition, random forest can work well with smaller datasets compared to other algorithms. Furthermore, random forest can deal with missing data.

However, one main problem with random forest is that visualizing the model is impossible. This is because we are taking many trees rather than one. This can make it difficult for people to interpret and understand the results without the support of the visualization. Lastly, if the model is deep the computational cost can become too much.

Data Preparation

We are going to use the DoctorAUS dataset to predict gender based on the variables that are in the dataset. If you want to know more about the variables in this dataset you can use the following code below.

```
data('DoctorAUS', show_doc=True)
```

For our purposes, we will load the code and take a brief look at the type of variables in the dataset.

```
In [13]: df = data('DoctorAUS')
    ...: df.info()
<class 'pandas.core.frame.DataFrame'>
Int64Index: 5190 entries, 1 to 5190
Data columns (total 15 columns):
sex          5190 non-null int64
age          5190 non-null float64
income       5190 non-null float64
insurance    5190 non-null object
illness      5190 non-null int64
actdays      5190 non-null int64
hscore       5190 non-null int64
chcond       5190 non-null object
doctorco     5190 non-null int64
nondocco     5190 non-null int64
hospadmi     5190 non-null int64
hospdays     5190 non-null int64
medecine     5190 non-null int64
prescrib     5190 non-null int64
nonpresc     5190 non-null int64
dtypes: float64(2), int64(11), object(2)
memory usage: 648.8+ KB
```

Most of the variables are fine however, we need to remove the following

when we create the X dataset.

- `sex` is the dependent variable so it must be removed when we make the independent variable dataset.

- `insurance` and `chcond` are categorical variables with text. They need to be converted to dummy variables in order for the algorithm to work. However, for our purposes this is unnecessarily complicated so we will remove them instead.

The code for all of this is below.

```
X=df.drop(['sex','insurance', 'chcond'], axis = 1)
```

We can now work to create the y dataset. We are going to recode it so that a 1 will mean female and a 0 male. Then we will create our y dataset and determine how many male and females in our dataset. Below is the code.

```
In [19]: df['sex']=df.sex.replace(1,'female')
    ...: df['sex']=df.sex.replace(0,'male')
    ...: y=df['sex']
    ...: y.value_counts()
    ...:
Out[19]:
female    2702
male      2488
Name: sex, dtype: int64
```

You can see how many males and females there are. If we add these two numbers together we can see that the total sample size is about 5200.The final step we will take for the data preparation is to create the train and test sets. This has been discussed in a prior chapter so we will just post the code below.

```
x_train, x_test, y_train, y_test

= train_test_split(X, y, test_size = 0.3, random_state=0)
```

We could explore the data more by looking at correlations, histograms and more. However, we will leave this to you to do. Now, we will turn our attention to model development.

Model Development

First, we will create our model. The code is mostly the same. The method that is used is different (`sk.RandomForestClassifier`) but the concept is the same. The argument `n_estimators` tells python to create 100 trees. Below is the code

```
clf=sk.RandomForestClassifier(n_estimators=100)
clf=clf.fit(x_train,y_train)
```

Now we can check the results. First, we will look at the variable importance. Again, this code is review and it is found below with the output.

```
In [22]: results=dict(zip(x_train.columns,
    ...:                  np.round(clf.feature_importances_,2)))
    ...: sorted(results.items(),
    ...:        key=lambda x: x[1],reverse=True)
Out[22]:
[('income', 0.22),
 ('age', 0.2),
 ('illness', 0.11),
 ('hscore', 0.11),
 ('prescrib', 0.09),
 ('medecine', 0.06),
 ('actdays', 0.04),
 ('doctorco', 0.04),
 ('hospdays', 0.04),
 ('nondocco', 0.03),
 ('nonpresc', 0.03),
 ('hospadmi', 0.02)]
```

You can see the results above but now we are going to take this process to the next level. We are going to make a bar graph of the results. This is done as follows.

1. Use the function `pd.Series`.

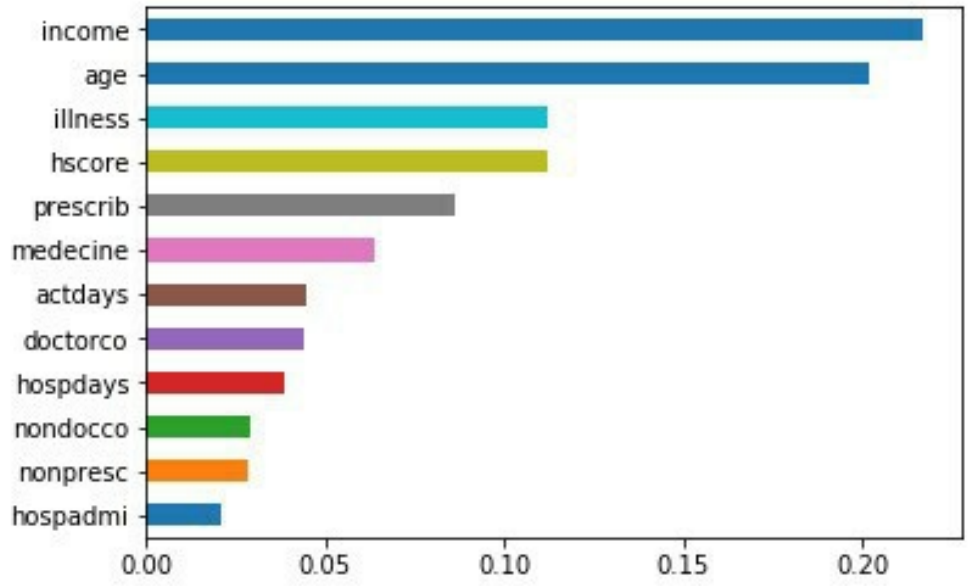
2. Inside this function you use the `clf` object with the `variable_importance` method and add the columns of the x dataset using the `.columns` method.

3. Next, you use the `sort_values` method so all the values are in order.

4. Last, save all this as an object called `model_ranks` and use the plot function to make the visual.

The code and visual is below.

```
model_ranks=pd.Series(clf.feature_importances_,
                      index=x_train.columns).sort_values(ascending=True)
ax=model_ranks.plot(kind='barh')
```



Income and age are the strongest predictors. This makes since as there is a strong difference in the salaries of men and women. In addition, there is a strong difference in the life expectancy of men and women.

We could have made this visual in chapter 1 but it is better to avoid teaching everything the first time people are exposed to it. In the code below, we take a look at the matrix and accuracy of our model when using the training data. All this code is review and will not be explained again.

```
In [34]: y_pred_train=clf.predict(x_train)
    ...: print(pd.crosstab(y_train,y_pred_train))
    ...: round(metrics.accuracy_score(y_train,y_pred_train),2)
col_0   female  male
sex
female    1748   154
male       107  1624
Out[34]: 0.93
```

This model looks pretty good with an accuracy of 93%. Let’s see how we do with the test data.

Model Testing

There is not much to explain as the code is the same as in chapter 1. Below is the code and results for the model testing.

```
In [35]: y_pred=clf.predict(x_test)
    ...: print(pd.crosstab(y_test,y_pred))
    ...: round(metrics.accuracy_score(y_test,y_pred),2)
col_0   female  male
sex
female     565   235
male       295   462
Out[35]: 0.66
```

There appears to be some problems with the model. To have our accuracy fall from 93% to 66% is perhaps too large of a drop. However, for our purposes we have completed all of our goals for this chapter.

Conclusion

Random forest allows you to mitigate the weaknesses of decisions trees through the use of many trees. Although you lose the ability to visualize the results you do reduce the potential for overfitting. It is always important to remember that when to use an algorithm depends more on your goals then anything else.

Chapter 4: Random Forest-Numeric Prediction

In this chapter, you will learn the following...

- Explain the characteristics of random forest regression.
- Explain the pros and cons of random forest regression.
- Develop a model using random forest regression.
- How to interpret the results of random forest regression.

Introduction

Random forest regression is used when the dependent variable is continuous in nature. The same concepts apply here as in the decision tree chapter on numeric prediction. The main difference in this chapter is that we are using random forest to make many decision trees rather than only one decision tree.

All of the strengths and weaknesses of random forest in general applies to regression. As such, the pros and cons will not be repeated here. In terms of our goal, we want to predict a person's income based on the other variables in the `DoctorAUS` dataset, which is the same dataset we used in the previous chapter.

Data Preparation

We will begin by loading our dataset and determining what variables we have. Below is the code.

```
In [52]: df = data('DoctorAUS')
    ...: df.info()
<class 'pandas.core.frame.DataFrame'>
Int64Index: 5190 entries, 1 to 5190
Data columns (total 15 columns):
sex          5190 non-null int64
age          5190 non-null float64
income       5190 non-null float64
insurance    5190 non-null object
illness      5190 non-null int64
actdays      5190 non-null int64
hscore       5190 non-null int64
chcond       5190 non-null object
doctorco     5190 non-null int64
nondocco     5190 non-null int64
hospadmi     5190 non-null int64
hospdays     5190 non-null int64
medecine     5190 non-null int64
prescrib     5190 non-null int64
nonpresc     5190 non-null int64
dtypes: float64(2), int64(11), object(2)
memory usage: 648.8+ KB
```

We need to drop the same text variables as last time (`insurance & chcond`). In addition, we need to remove our dependent variable income from the independent variable dataset because it cannot be dependent and independent at the same time. When this is done we can make our dataset for the independent variables and the dependent variable. Below is the code for completing this step.

```
X=df.drop(['insurance', 'chcond','income'], axis = 1)

y=df['income']
```

The final step of the data preparation process is to create the train and test sets. Below is the code for this.

```
x_train, x_test, y_train, y_test = train_test_split(X, y,
test_size=0.3, random_state=0)
```

We will now proceed to the model development

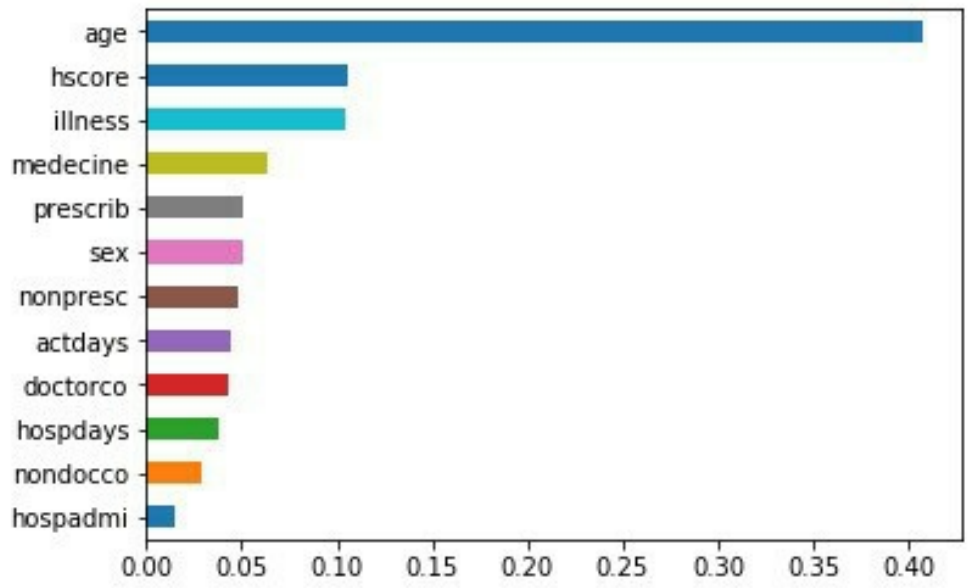
Model Development

First, we will begin by creating the model. There is nothing new in the code below except we are using the `sk.RandomForestRegressor` method.

```
reg=sk.RandomForestRegressor(n_estimators=1000)

reg=reg.fit(x_train,y_train)
```

In the code below, we will look at the variable importance. We will not print the actual values. Instead, we will only print the bar graph, which we learned about in a previous chapter.

```
model_ranks=pd.Series(reg.feature_importances_,
                      index=x_train.columns).sort_values(ascending=True)
ax=model_ranks.plot(kind='barh')
```



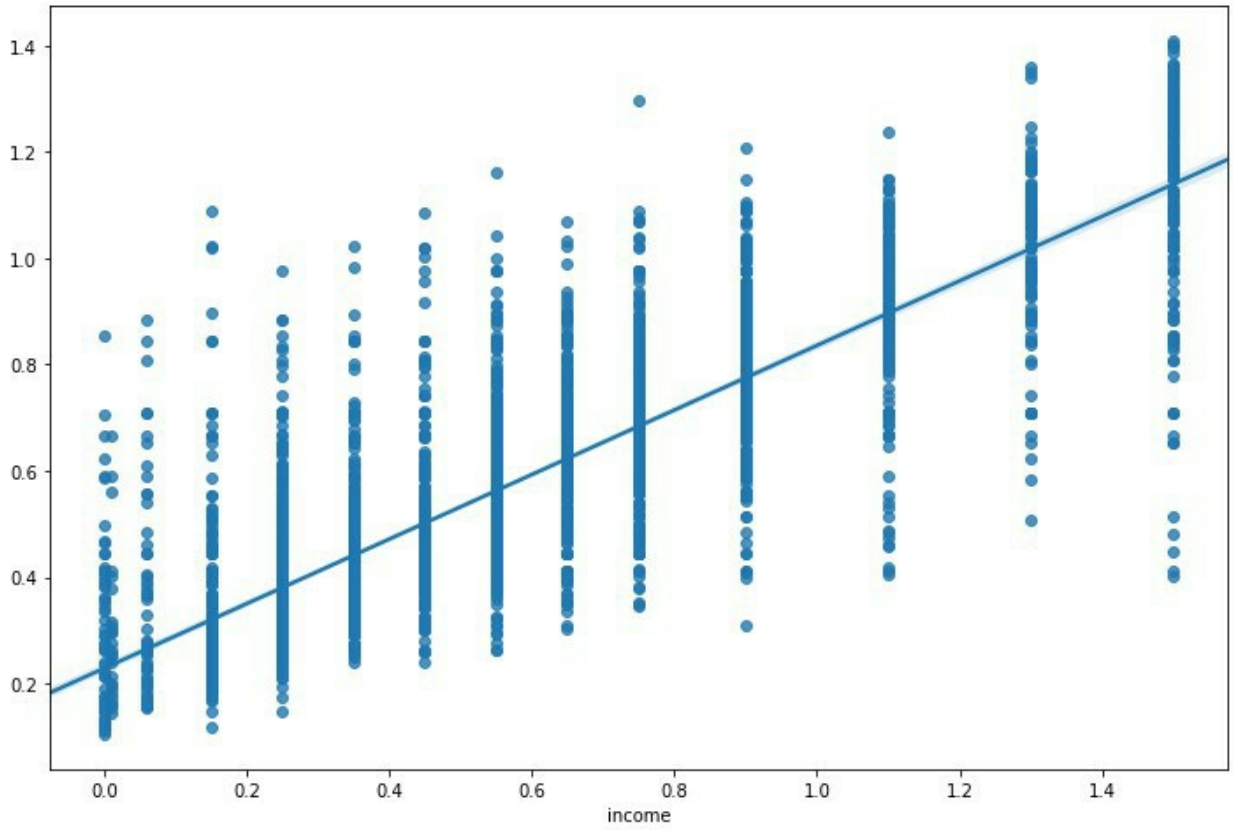
You can clearly see that age is by far the most important variable in this model. This make sense as we get older we often make more money. Next, we will see how strong the prediction is by predicting with our model and looking at the correlation.

```
y_pred_train=reg.predict(x_train)

np.corrcoef(y_pred_train,y_train)
```

```
Out[61]:

0.83
```

The correlation appears to be strong. Below is the plot.

```
fig, ax = plt.subplots(figsize=(12,8))

sns.regplot(y_train, y_pred_train)
```



Well, it is linear somewhat but not as strongly as the numbers would suggest. Lastly, we will calculate the mean squared error.

```
round(mean_squared_error(y_train, y_pred_train),2)

Out[64]: 0.04
```

Remember, that this value lacks meaning unless it is compare to another model. For now, we will see how our model does with the test data.

Model Testing

There is little to say. We will first predict with the test data. Then, we will find the correlation and make the scatterplot. Lastly, we will calculate the MSE.

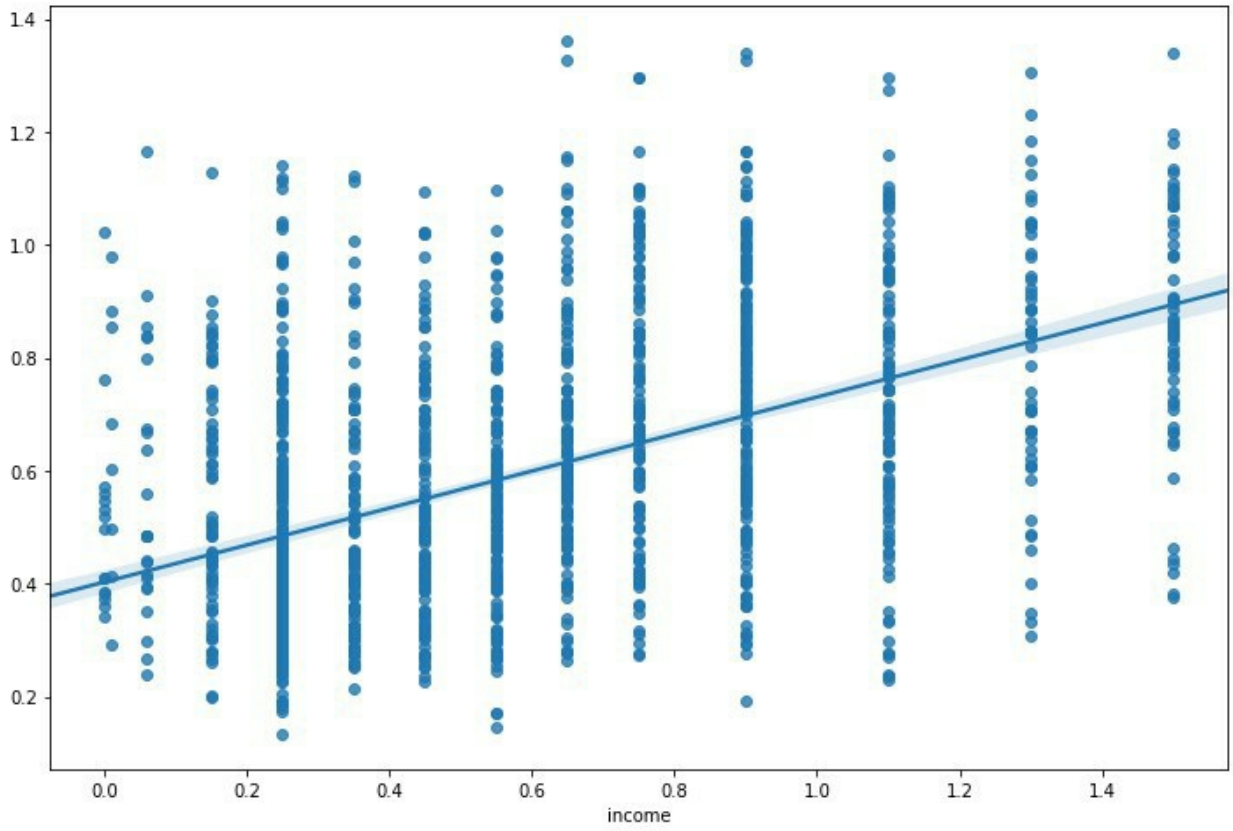
```
y_pred=reg.predict(x_test)
```

```
np.corrcoef(y_pred,y_test) Out[65]: 0.48
```

We can see that the correlation has gone down quite a bit. Let's look at the scatterplot.

```
fig, ax = plt.subplots(figsize=(12,8))
```

```
sns.regplot(y_test, y_pred)
```



Looks rather messy. Let's check the MSE

```
round(mean_squared_error(y_train, y_pred_train),2)
```

```
Out[64]: 0.11
```

This is similar to our training data results. However, there may be problems with our model if we tried to use it in different situations.

Conclusion

Random forest regression is a chance to use random forest for continuous dependent variables. This chapter simply demonstrated how to do it. The steps are mostly the same, however, the tools used for assessing the model are somewhat different. If you do not need a visual than random forest might be a tool for you.

Chapter 5: K Nearest Neighbor-Classification & Numeric Prediction

In this chapter, you will learn the following...

- Explain the characteristics of K Nearest Neighbor-Classification.
- Explain the pros and cons of K Nearest Neighbor-Classification.
- Develop a model using K Nearest Neighbor-Classification.
- How to interpret the results of K Nearest Neighbor-Classification.

Introduction

K Nearest Neighbor (KNN) is an algorithm that uses known examples that are near each other to predict the class of an unknown example that is nearby as well. To give an illustration of this, imagine that you are at a school inside a classroom. There are 45 students in the classroom all between the ages of 12-13. If another young person walks into the room around the same age you would probably predict they are a student as well. In other words, you are taking what you know about the classroom and the characteristics of the students who are similar to each other to predict a new unknown person who is also similar in appearance.

KNN uses the variables or features of the dataset to map a multidimensional space in which it places all the examples of the sample. Neighbors are calculated using most commonly euclidean distance, which is a measure of the straight line distance between points. The lower the euclidean distance the closer neighbors are to each other.

The "K" in KNN is the number of neighbors to use when trying to predict an unknown example. For classification, the K is usually an odd number because the neighbors "vote" when predicting the classification of an example and the winner is a simple majority. For example, if I use 3 neighbors to predict an unknown example there will always be a majority when each neighbor votes. If the goal is numeric prediction than the average of the neighbors is used to determine the value of the unknown example. It is also important to mention that as the K increases the classification can become much more complicated or non-linear in nature.

In terms of pros and cons, KNN is good with non-linear data. In addition, KNN is a non-parametric algorithm, in which it does not make any assumptions about the data. Therefore, this algorithm can be highly flexible while maintaining a rather simplistic model that is easy to understand.

Cons include the fact that KNN is a lazy learning algorithm. This means that there is no abstraction of the data. This leads to a problem that the work is done but nobody understands it as there is a lack of explanation of the relationship between the variables. This is related to the strength of being

non-parametric. By lacking parameters there are no insights into the characteristics of the data. In addition, KNN will struggle with large datasets as everything is stored in memory when computation is taking place.

This chapter will be slightly different from the other chapters in that there will be one example of classification and numeric prediction in the same chapter rather than separating them. This was done because the coding is no that complicated in comparison to other chapters in the book.

Classification

We are going to use the `turnout` dataset to predict if someone voted or not based on the other variables in the dataset.

Data Preparation

Below are the libraries that will be used.

```
from pydataset import data
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn import metrics
```

We will now load our dataset and take a look at the variables.

```
In [12]: df=data("turnout")
    ...: df.info()
<class 'pandas.core.frame.DataFrame'>
Int64Index: 2000 entries, 1 to 2000
Data columns (total 5 columns):
race       2000 non-null object
age        2000 non-null int64
educate    2000 non-null float64
income     2000 non-null float64
vote       2000 non-null int64
dtypes: float64(2), int64(2), object(1)
memory usage: 93.8+ KB
```

The `race` variable will need to be removed because it is text based and not continuous. We could recode it but the majority of the examples are classified as "White" in the variable so it is adding little value if we were to keep it. You can confirm this with the following code.

```
df['race'].value_counts()
```

With the code below, we will drop the race variable and take a look at the first few entries of the dataset using the `.head()` method.

```
In [16]: df=df.drop('race',1)
    ...: df.head()
Out[16]:
   age  educate  income  vote
1   60     14.0  3.3458     1
2   51     10.0  1.8561     0
3   24     12.0  0.6304     0
4   38      8.0  3.4183     1
5   25     12.0  2.7852     1
```

KNN is scale sensitive. This means that variables that have a larger scale will have a stronger influence on the predictions. For example, the age variable is on a much larger scale than the other variables which will give it a stronger influence on the prediction. Therefore, we will scale the variables. Once we scale we will look at the dataset one more time.

```
In [17]: df = (df - df.min()) / (df.max() - df.min())
    ...: df.head()
    ...:
Out[17]:
        age   educate    income  vote
1  0.551282  0.736842  0.224179   1.0
2  0.435897  0.526316  0.124364   0.0
3  0.089744  0.631579  0.042239   0.0
4  0.269231  0.421053  0.229036   1.0
5  0.102564  0.631579  0.186617   1.0
```

We will now prepare are X and y datasets. First, we will create the X dataset for the independent variables. Second, we will recode the dependent variable to indicate "yes" or "no" that the individual did not vote. Third, we will create

our y dataset for our dependent variable. Last, we will look at a table to see how many people voted and did not vote. The code is below.

```
In [18]: X=df[['age','educate','income']]
    ...: df['vote']=df.vote.replace(1,'yes')
    ...: df['vote']=df.vote.replace(0,'no')
    ...: y=df['vote']
    ...: y.value_counts()
Out[18]:
yes    1492
no      508
Name: vote, dtype: int64
```

Finally, we will create our train and test datasets.

```
X_train,X_test,y_train,y_test=train_test_split(X,
y,test_size=.3,random_state=0)
```

We can now move to model development.

Model Development

The code is rather simple here. We create an instance of our classifier than we create the model. One thing to mention is that for now, the K is set to 7. Again, there are ways to pick the best K that will be discussed in a future chapter

```
clf=KNeighborsClassifier(7)
clf.fit(X_train,y_train)
```

In the code below, we predict, create the confusion matrix, and find the accuracy.

```
In [21]: y_pred_train=clf.predict(X_train)
    ...: print(pd.crosstab(y_train,y_pred_train))
    ...: round(metrics.accuracy_score(y_train,y_pred_train),2)
col_0   no  yes
vote
no     136  218
yes     80  966
Out[21]: 0.79
```

You can see the overall accuracy is at almost 80%. We will now see how our model does with the test data.

Model Testing

The code and results are below.

```
In [22]: y_pred_test=clf.predict(X_test)
    ...: print(pd.crosstab(y_test,y_pred_test))
    ...: round(metrics.accuracy_score(y_test,y_pred_test),2)
    ...:
col_0  no  yes
vote
no     37  117
yes    48  398
Out[22]: 0.72
```

The accuracy fell as expected but the model seems to be fairly consistent. We will now turn our attention to using KNN for numeric prediction.

Regression

Our goal this time will be to predict income using the other variables.

Data Preparation

Below are the libraries followed by the rest of the data preparation. We have already discussed this so there is no reason to repeat it here.

```
from pydataset import data
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsRegressor
import numpy as np
from sklearn.metrics import mean_squared_error
import matplotlib.pyplot as plt
import seaborn as sns


df=data("turnout")
df=df.drop('race',1)
df = (df - df.min()) / (df.max() - df.min())

X=df[['age','educate','vote']]
y=df['income']

X_train,X_test,y_train,y_test=train_test_split(X,y,test_size=.3,random_state=0)
```

We can now move to model development.

Model Development

In the code below we train our model. Notice how the method is the regressor rather than the classifier.

```
reg=KNeighborsRegressor(7)

reg.fit(X_train,y_train)
```

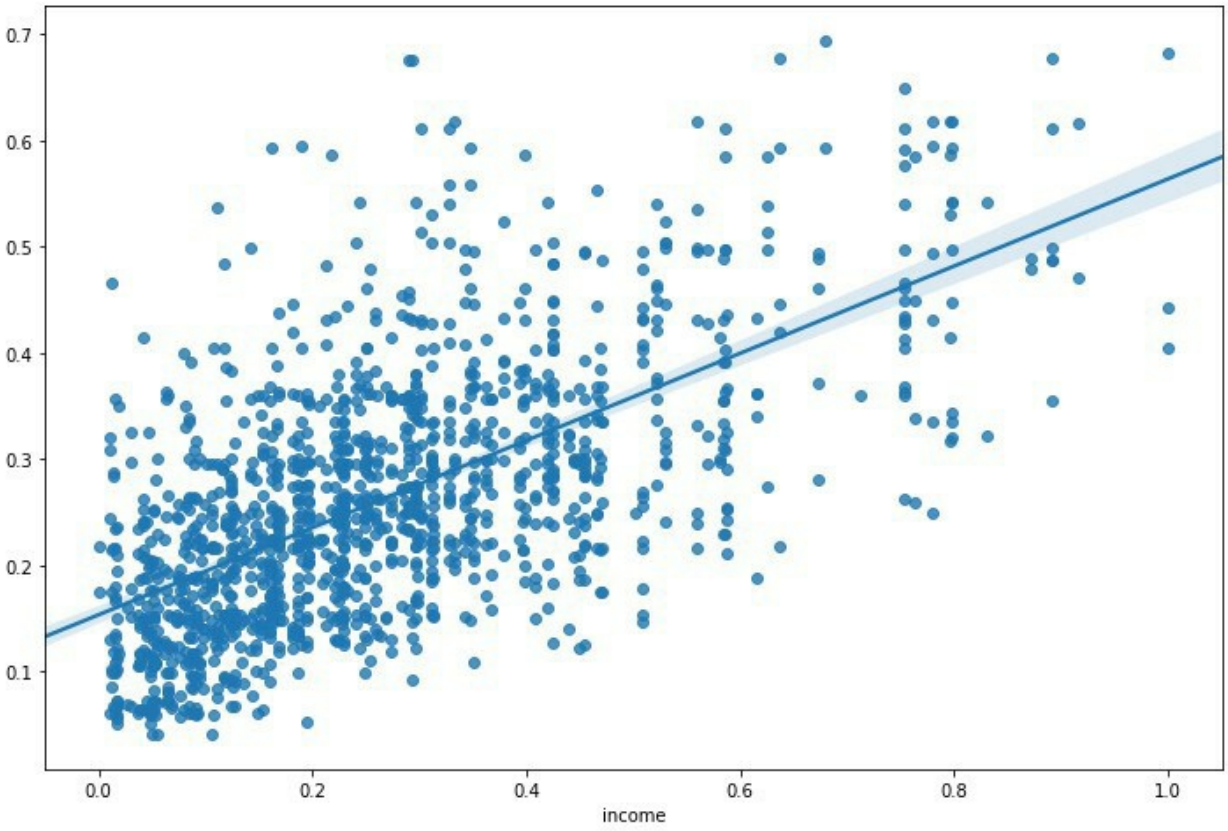
In the code below, we make the predictions and calculate the correlation between the actual and predicted values.

```
y_pred_train=reg.predict(X_train)

np.corrcoef(y_pred_train,y_train)
```

```
0.63
```

The correlation of 0.63 seems strong. Below is the scatterplot.

```
fig, ax = plt.subplots(figsize=(12,8))
sns.regplot(y_train, y_pred_train)
```



This looks good. Finally, we calculate the MSE.

```
round(mean_squared_error(y_train, y_pred_train),5)
0.02108
```

You can see the MSE fr yourself. Let’s move to model testing.

Model Testing

Below is the code with the results.

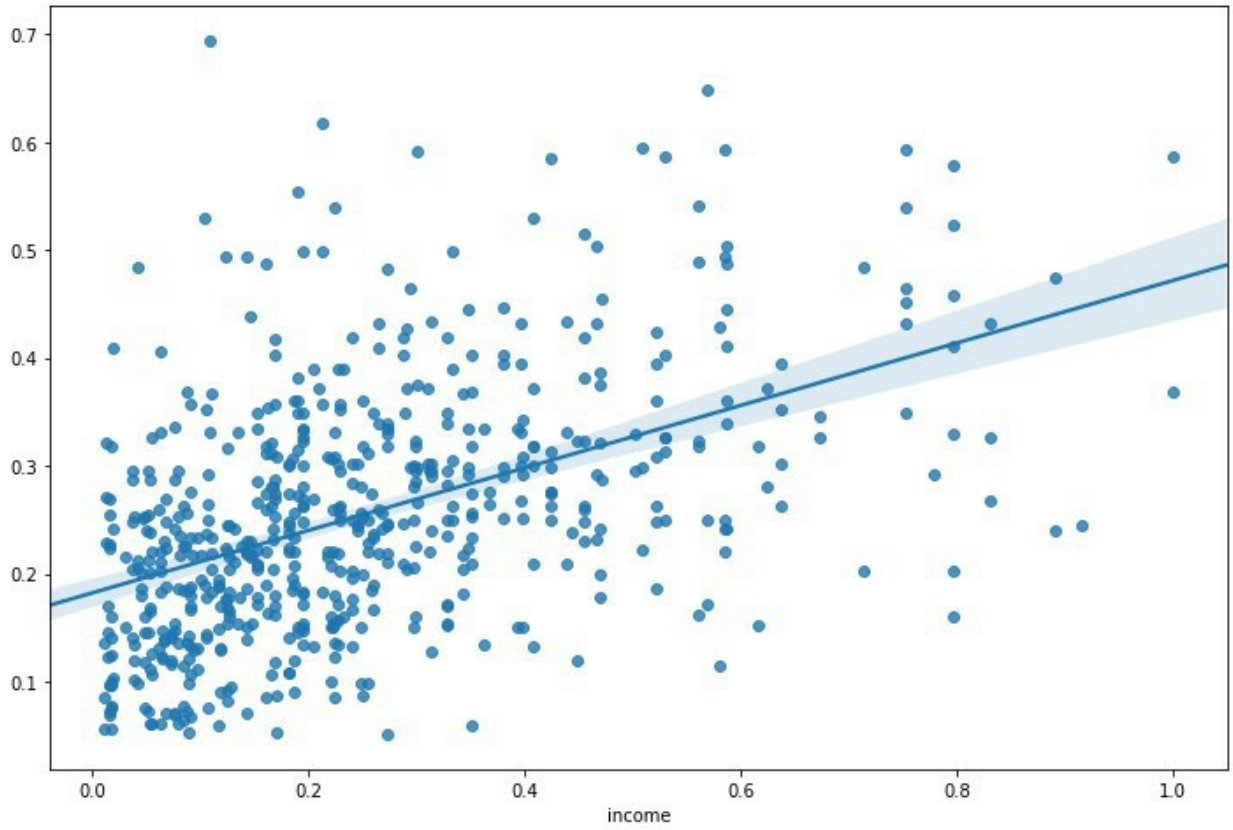
```
y_pred_test=reg.predict(X_test)
```

```
np.corrcoef(y_pred_test,y_test)
```

```
0.48
```

There is the usual drop off in the correlation. Below is the scatterplot.

```
fig, ax = plt.subplots(figsize=(12,8))
sns.regplot(y_test, y_pred_test)
```



Finally, the MSE

```
round(mean_squared_error(y_test, y_pred_test),5)
```

```
0.02888
```

Not much difference. Perhaps the model generalizes.

Conclusion

This chapter provide examples of how to use KNN for both classification and numeric prediction. These tools are yet another option to be considered when doing quantitative analysis. As with other algorithms discussed so far KNN has its strengths and weaknesses. However, there are times when this tool may be the best option.

Chapter 6: Support Vector Machines-Classification

In this chapter, you will learn the following...

- Explain the characteristics of support vector machines classification.
- Explain the pros and cons of support vector machines classification.
- Develop a model using support vector machines classification.
- How to interpret the results of support vector machines classification.

Introduction

Support vector machines is a combination of the K-nearest neighbor algorithm and linear regression. For k-nearest neighbor, examples that are close to an unknown sample are used to determine the unknown sample's class. For linear regression, a line is drawn to try and separate different classes in the dataset. This combination of grouping and line development leads to something called a "hyperplane" which separates the examples in the dataset. The hyperplane is what creates the greatest distance between the different classes. The examples that are on the border of the hyperplane are called the support vectors. The vectors are what create or support the boundaries between the classes. Figure 6.1 provides a picture of what we are discussing.

In the example, we are trying to separate the x's from the o's. The x's and the o's on the line are the support vectors. They are used to make the boundaries. In the example, you can see that two o's are used and one x. The distance between these support vectors is called the margin. The vectors plus the margin is the hyperplane. The goal is to always have the widest margin possible.

There are several distinct traits of SVM that have not been addressed before. First, it is preferred to draw straight lines but if this is not the case the algorithm can adjust through the use of an slack variable which facilities error and or mis-classifications. This value can be adjusted and we will learn how in a latter chapter.

Second, SVM can also use what is called a kernel trick. This involves the creation of additional variables or features to improve the accuracy by translating the data into higher dimensions. This allows for better hyperplanes to be developed. How this works depends on the kernel that is employed when developing the model. These kernels are particularly useful when the data is non-linear. The details of this is beyond the scope of the text but is something to be aware of.

As with all algorithms, SVM has its strengths and weaknesses. For strengths, SVM is highly flexibility, can handle messy data, and can be accurate despite

not being able to determine how the algorithm obtain the results.

For cons, SVM is an example of a black box method. This means that it is not really clear what the algorithm did to get the answer. This can make it impossible to explain to a non-mathematical person. It is also hard to determine which kernel to use and when in many circumstances.

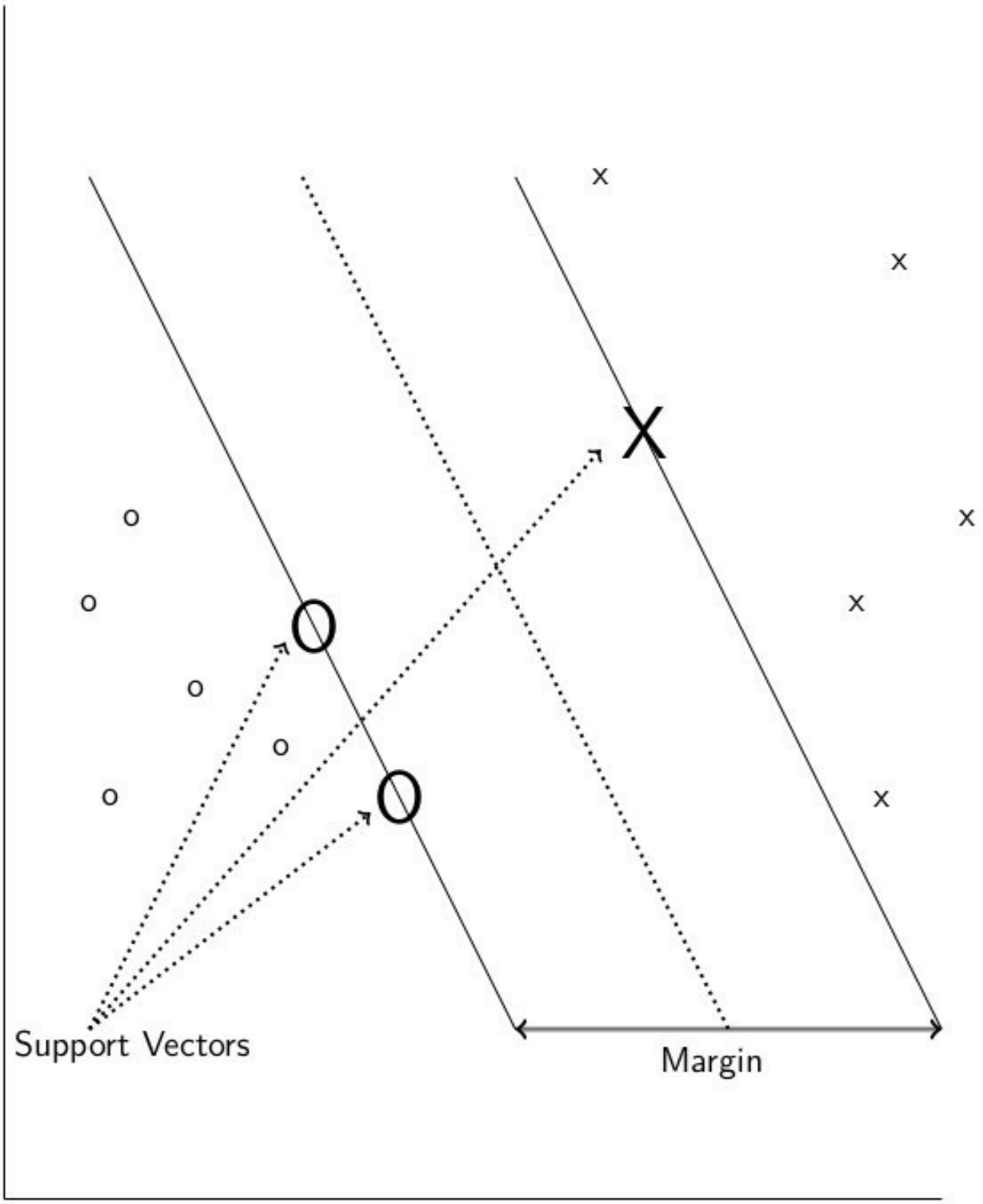


Figure 6.1: Support Vectors & Hyperplane

Data Preparation

Our goal is to predict whether someone has a mortgage or not based on the other variables in the `Workinghours` dataset. Below is the initial code to load the libraries that we will use.

```
import numpy as np
from pydataset import data
from sklearn import svm
from sklearn.metrics import mean_squared_error
from sklearn import model_selection
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd
```

Next, will do the following.

1. Load the dataset
2. Remove any NA
3. Use the `.info()` function to see what type of variables we have

Below is the code and results.

```
In [2]: df=pd.DataFrame(data('Workinghours'))
   ...: df=df.dropna()
   ...: df.info()
<class 'pandas.core.frame.DataFrame'>
Int64Index: 3382 entries, 1 to 3382
Data columns (total 12 columns):
hours         3382 non-null int64
income        3382 non-null int64
age           3382 non-null int64
education     3382 non-null int64
child5        3382 non-null int64
child13       3382 non-null int64
child17       3382 non-null int64
nonwhite      3382 non-null int64
owned         3382 non-null int64
mortgage      3382 non-null int64
occupation    3382 non-null object
unemp         3382 non-null int64
dtypes: int64(11), object(1)
memory usage: 343.5+ KB
```

Looking at the variables we need to remove the `occupation` variable because it is text based. In addition, we have several child variables we need to combine. `child5` is number of children under five, `child13` is number of children under 13 but above 5, and `child17` is the number of children above 13 but below 17. Personally, a child is a child so we are going to combine these three variables. Below is the code for removing `occupation` and combing the three child variables.

```
df=df.drop(['occupation'],1)

df['all_kids']=df['child5']+df['child13']+df['child17']
```

Now, we need to rescale the data. Just like with K nearest neighbor SVM can be sensitive to scale. Therefore, we will rescale the dataset so that values fall between 0 and 1. This is done by take the values of the data subtracting them from the minimum and dividing by the values of the dataset subtracted from the max and minimum value. The code is below.

```
df = (df - df.min()) / (df.max() - df.min())
```

For interpretation purposes, it would be good to change the dependent variable `mortgage` to text. We've done this before so the code is below.

```
df['mortgage']=df.mortgage.replace(1,'yes')
```

```
df['mortgage']=df.mortgage.replace(0,'no')
```

We will now create are X and y datasets and count the values in the y variable. The code is below.

```
In [79]: X=df[['hours','income','age','education','all_kids',
    ...:        'nonwhite','owned','unemp']]
    ...: y=df['mortgage']
    ...: y.value_counts()
    ...:
Out[79]:
yes    1785
no     1597
Name: mortgage, dtype: int64
```

The y variable is fairly balanced between yes and no. Generally, you want to avoid extremes as it helps the prediction.

Lastly, we create our train and test sets.

```
In [8]: X_train,X_test,y_train,y_test=model_selection.train_test_split(X,y,test_size=.
3,random_state=1)
```

We can now move on to the model development phase.

Model Development

We are going to do things slightly differently during the model development phase compared to prior chapters. As mentioned previously, svm uses a kernel trick to fit data. As such, there are multiple kernels that can be used to model the data. Therefore, we are going to use two different kernels and compare their performances. The two kernels are the linear kernel which tries to draw a straight line to separate groups and the radial basis function kernel, which assumes a non-linear boundary between the classes.

Both of these kernels have hyper-parameters, which are arguments that must be set manually before fitting can take place. For now, we will just set these values randomly but later we will learn how to find the best values based on the data.

Below is the code for the linear kernel. Most of this is review so there will be little explanation. What is new is setting the C value to 1 inside the `LinearSVC` method. This C value is the hyper-parameter.C stands for cost function and it is the "price" for mis-classification. We will fit the model, make the predictions and then check the accuracy.

```
In [11]: h1=svm.LinearSVC(C=1)
    ...: h1.fit(X_train,y_train)
    ...: y_pred_train=h1.predict(X_train)
    ...: print(pd.crosstab(y_train,y_pred_train))
    ...: round((1164+897)/(897+242+64+1164),2)
    ...:
col_0       no   yes
mortgage
no         897   242
yes         64  1164
Out[11]: 0.87
```

The accuracy is 87%, which is considered good in many context. We calculate the accuracy manually because the method we used in the past does not like the results that come from the svm function. We will not assess the

model with the test data because we are only suppose to use the test data once. For now, we will see if the rbf kernel can outperform the linear kernel.

For the rbf kernel, we will set three hyper-parameters, which are the degree, gamma, and C. Degree relates to the use of polynomial features, gamma is harder to explain but has to do with the separation of the samples vertically when making higher dimensions. The higher the gamma means a softer bump while a smaller gamma creates a stronger bump. The C hyper-parameter was already explained.

```
In [13]: h2=svm.SVC(kernel='rbf',degree=3,gamma=0.001,C=1.0)
    ...: h2.fit(X_train,y_train)
    ...: y_pred_train=h2.predict(X_train)
    ...: print(pd.crosstab(y_train,y_pred_train))
    ...: round((786+1228)/(786+1228+0+353),2)
col_0       no   yes
mortgage
no         786   353
yes          0  1228
Out[13]: 0.85
```

You can see that our accuracy went down a little bit. This indicates that rbf is not the best performer. Therefore, we will test our model using the linear kernel.

Model Testing

The results for the model testing is below. There is little to explain as this has been done in previous chapters.

```
In [17]: y_pred_test=h1.predict(X_test)
    ...: print(pd.crosstab(y_test,y_pred_test))
    ...: round((329+534)/(329+534+23+129),2)
col_0       no  yes
mortgage
no         329  129
yes         23  534
Out[17]: 0.85
```

The accuracy went down slightly, which is to be expected. In all, this model does a fairly good job at predicting if someone has a mortgage or not. The main problem is we really don't know how the algorithm did this. With random forest and decision trees we can at least look at the variable importance but this is missing in this situation.

Conclusion

Support vector machines provide you with a powerful tool for getting results. However, if you want to understand what is going on this may not be the best algorithm to consider as SVM is good at predicting but not explaining how it calculated the predictions.

Chapter 7: Support Vector Machines-Regression

In this chapter, you will learn the following...

- Explain the characteristics of support vector machines regression.
- Explain the pros and cons of support vector machines regression.
- Develop a model using support vector machines regression.
- How to interpret the results of support vector machines regression.

Introduction

We are now going to revisit SVM but this time from the prospective of numeric prediction. The concepts are mainly the same. We will use two different kernels in order to compare performance.

Data Preparation

We will use the `Workinghours` dataset again but this time to predict the education level of a person. Most of what we will do here is a review of the previous chapter. Here is a summary of what we will do.

1. Load the needed libraries
2. Load the dataset and drop any potential na's
3. Drop the occupation variable because it is a text variable
4. Combine the variables `child5, child13, & child17` into one variable called `all_kids`
5. Scale the data
6. Separate the independent variables from the dependent variable
7. Create the train and test datasets

The code for this is below.

1. Load the needed libraries

```
import numpy as np
from pydataset import data
from sklearn import svm
from sklearn.metrics import mean_squared_error
from sklearn import model_selection
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd
```

2. Load the dataset and drop any potential na's

```
df=pd.DataFrame(data('Workinghours'))
df=df.dropna()
```

3. Drop the occupation variable because it is a text variable

4. Combine the variables child5, child13, & child17 into one variable called all_kids

```
df=df.drop(['occupation'],1)
df['all_kids']=df['child5']+df['child13']+df['child17']
```

5. Scale the data

```
df = (df - df.min()) / (df.max() - df.min())
```

6. Separate the independent variables from the dependent variable

```
X=df[['hours','age','income','all_kids',
      'nonwhite','owned','unemp']]
y=df['education']
```

7. Create the train and test datasets

```
In [85]: X_train,X_test,y_train,y_test=model_selection.train_test_split(X,y,test_size=.
3,random_state=1)
```

We can now move to model development.

Model Development

We are going to use two different kernels just as last time. It will be the same two kernels but this time we will be using them in the context of regression. Our first kernel will be the linear kernel. Below is the code for fitting the model and making the predictions.

```
h1=svm.LinearSVR(C=1.0, max_iter=20000)
h1.fit(X_train,y_train)
y_pred_train=h1.predict(X_train)
```

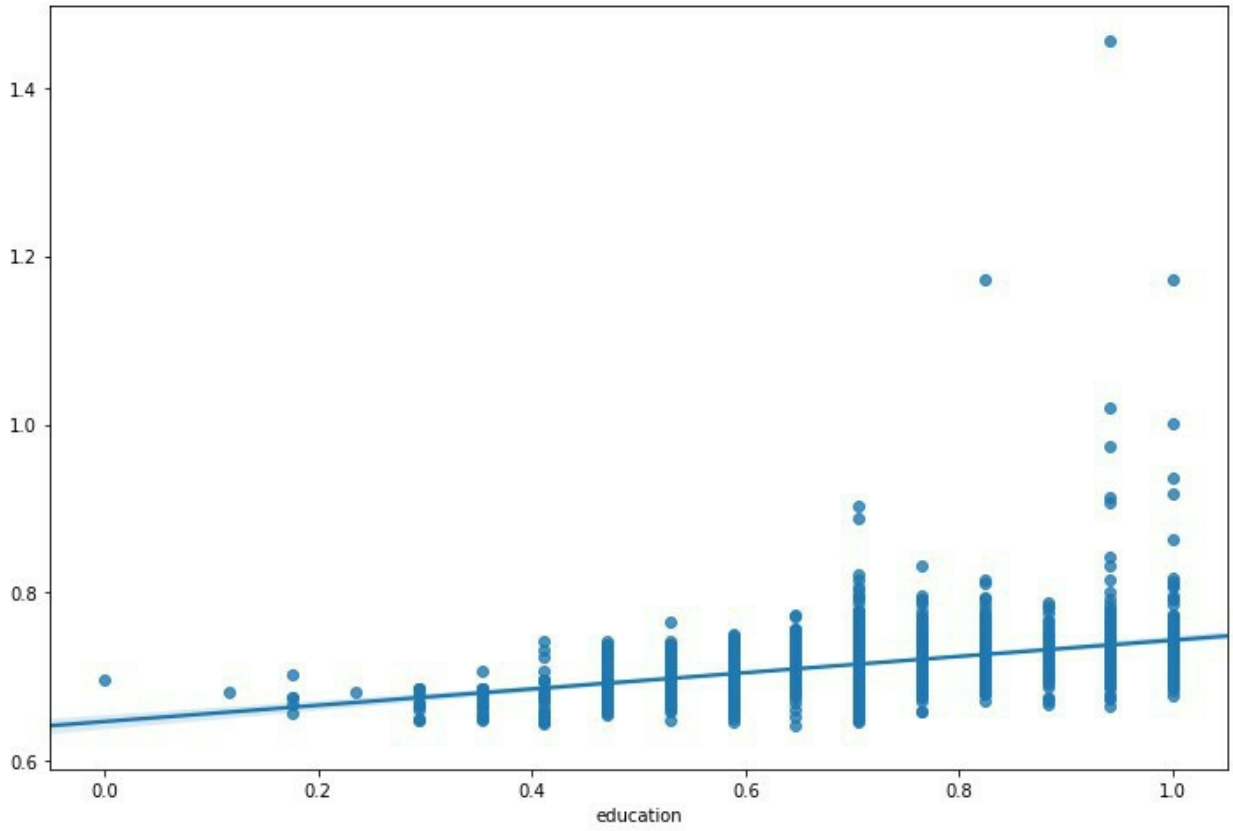
Next, we calculate the coefficients, make the scatterplot, and look at the mean squared error.

```
np.corrcoef(y_train,y_pred_train)
```

```
0.38
```

```
fig, ax = plt.subplots(figsize=(12,8))
```

```
sns.regplot(y_train, y_pred_train)
```



Finally the mse,

```
round(mean_squared_error(y_train, y_pred_train),5)
0.01789
```

The correlation is weak between the actual and predicted values. One of the main problems is that there are some strong outliers in the scatterplot. If this was for real we would want to identify and consider removing these outliers. The MSE is only useful for comparison and will be our metric that we use to see which kernel is better. We will now run the code for the second kernel, which is a radial basis function kernel (rbf). Below is the code.

```
h2=svm.SVR(kernel='rbf',degree=3,gamma=0.001,C=1.0)
```

h2.fit(X_train,y_train)

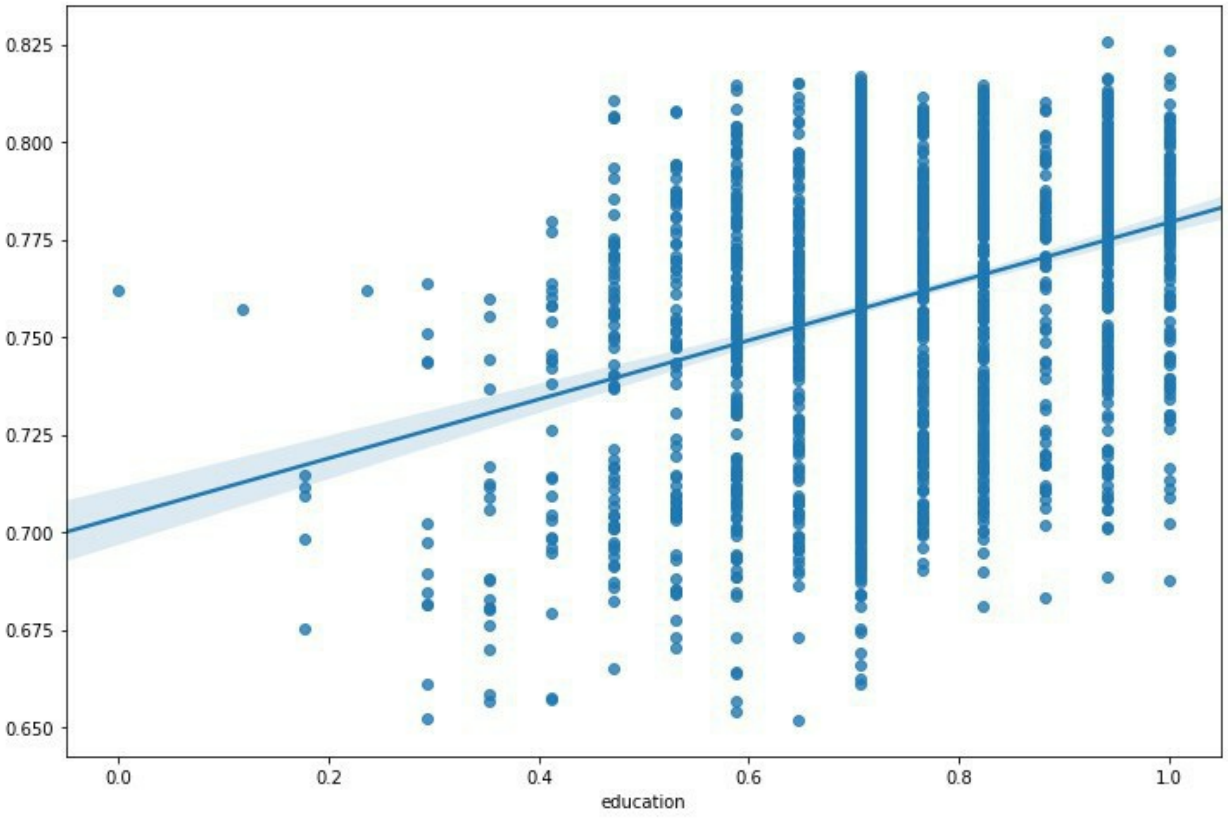
This kernel has additional hyperparameters, which are degree and gamma. These were discussed previously and the values are arbitrary. We will

discussed hyperparameter tuning in a later chapter. Next, we will predict and calculate the correlation between the actual and predicted values.

```
y_pred_train=h2.predict(X_train)

np.corrcoef(y_train,y_pred_train)

0.31
```

The correlation is slightly lower. Let's exam the scatterplot and calculate the MSE.

```
fig, ax = plt.subplots(figsize=(12,8)) sns.regplot(y_train,
y_pred_train) round(mean_squared_error(y_train, y_pred_train),5)

0.01878
```



The MSE is slightly higher than the previous model. There's little difference. For the test data, we will use the first model with the linear kernel.

Model Testing

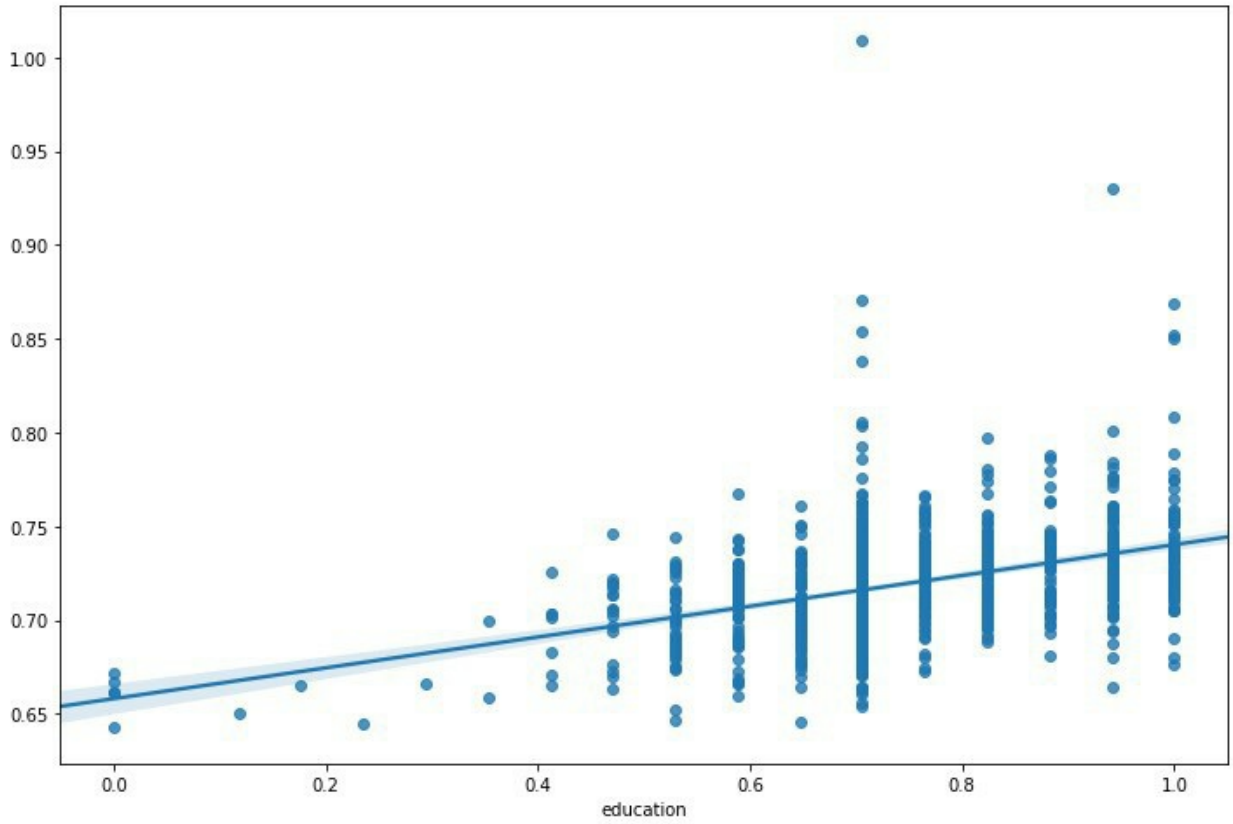
The code below shows the performance of the first model with the test data. First we make the predictions and calculate the correlation.

```
y_pred=h1.predict(X_test)
```

```
np.corrcoef(y_pred,y_test)
```

```
0.40
```

The correlation is actually a little better than the training data. Below is the scatterplot for visualization purposes.



Lastly, here is the MSE

```
round(mean_squared_error(y_test, y_pred),5)
```

```
0.01832
```

The MSE is almost the same. It appears that this model would generalize to

other data.

Conclusion

This chapter provided an example of how to use support vector machines for predicting numeric values. Many of the same ideas apply for numeric prediction as they did for classification. The names of the actual methods used are different but the concepts remain the same.

Chapter 8: Artificial Neural Networks

In this chapter, you will learn the following...

- Explain the characteristics of Artificial Neural Networks.
- Explain the pros and cons of Artificial Neural Networks.
- Develop a model using Artificial Neural Networks.
- How to interpret the results of Artificial Neural Networks.

Introduction

In this chapter, we are going to learn briefly about artificial neuron networks (ANN). We will learn how to classify with this algorithm and we will also look at numeric prediction with this algorithm.

Understanding Artificial Neural Networks

Artificial neurons were inspired at least in part by the actual behavior and functions of the neurons of the brain. In lay terms, as this is not a book on human biology, neurons send and receive signals inside the body. Information is received through dendrites, processed in the soma, and messages are sent via the axon terminal. Below is a picture of a neuron in figure 8.1.

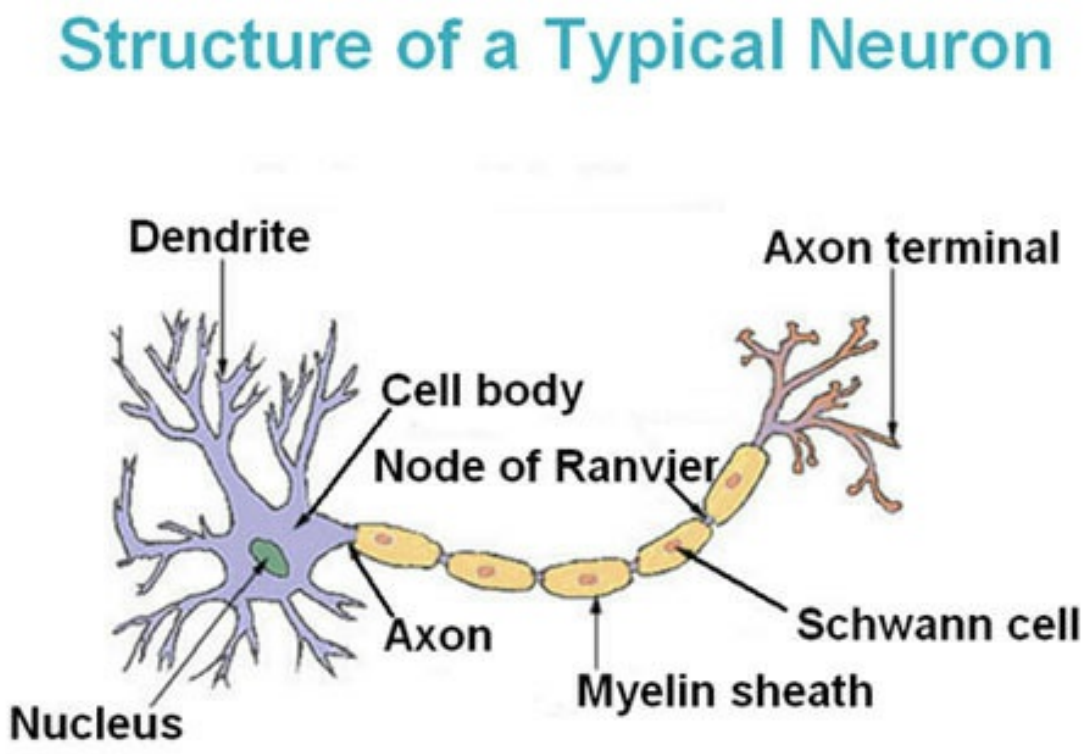


Figure 8.1: Neuron

An artificial neuron works in a similar manner. The input or independent variables can be consider the dendrites which provide information to the cell body where they are further transformed through some form of calculation. Furthermore, just as in regression, different dendrites or x variables can have different weights (w) that lead to the next part of the neuron. Next, the summation of the independent/x variables are passed to an activation function or the soma. There are different types of activation that can take place with different functions. Lastly, the information moves to the output or dependent variable y, which is similar to our axon terminal in a real neuron. This

process can then be repeated many times as information from one neuron to the other in what can become a highly complex process. Below is a picture of of one artificial neuron in figure 8.2.

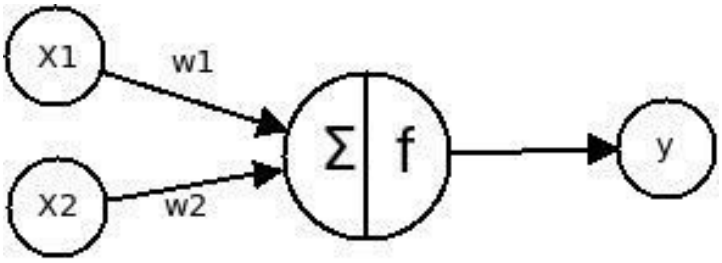


Figure 8.2: Artificial Neuron

If you compare the two pictures they are similar yet different. The artificial neuron mimics the mind in a way that has fascinated people for over decades. However, it is highly important to note that the ANN community is moving away from this analogy as being unsatisfactory. Despite this, it is good to be familiar with this classic explanation and is a useful comparison for people new to this conversation.

Activation Function (f)

The purpose of the activation function is to determine if there should even be an activation. Returning to the human body analogy, activation takes place when the nerve cell sends the message to the next cell and this only happens if there is a message to send. In other words, no stimulus no activation. If neurons were firing all the time it would be torturous for most people.

The same concept applies with artificial neurons. A signal will not be passed on unless it meets a minimum threshold. Several factors can effect the threshold such as how the artificial neuron is designed, the activation function chosen, and the hyperparameter settings the researcher provides.

There is a lot of mathematics involved in fully understanding ANN. For example, a person would need to be familiar with concepts associated with Calculus such as, derivatives, as well as vectorization. In addition, there is an idea known as backward and forward propagation that relates to the behavior of the ANN. As interesting as these topics are they are beyond the scope of

this text.

ANN Design

Currently, we have only talk about the characteristics of one artificial neuron. However, just as real neurons work together in a complex network so to do artificial neurons work together in a network. The makeup of an ANNs can vary greatly depending on the situation. Some models have more than one output variable as shown below in figure 8.3.

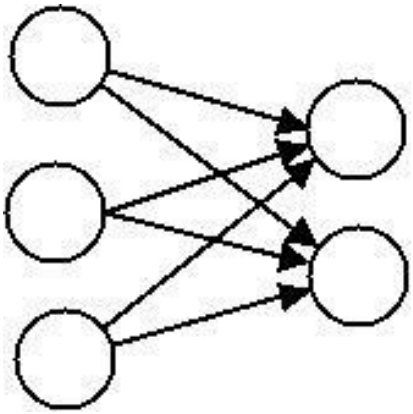


Figure 8.3: Two Output ANN

It is common for ANN models have what are called hidden layers. These are neurons that serve as both input and output variables. It is the researcher who determines the number of nodes in a hidden layer based on experience and other factors. In addition, the researcher also must indicate how many hidden layers to include in the model. Below is a visual example in figure 8.4.

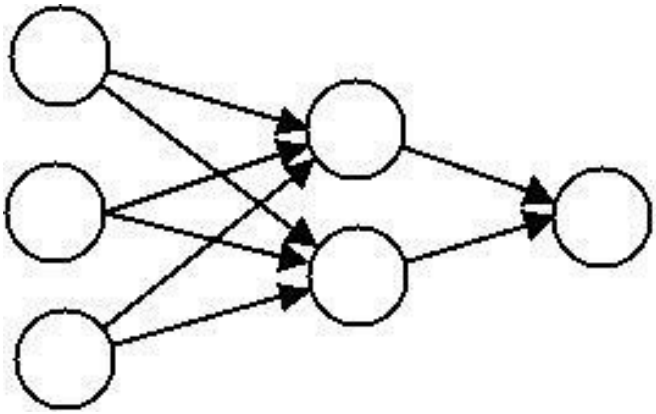


Figure 8.4: One Output ANN with Hidden Layer

When a model has several hidden layers it is often referred to as a deep learning network. How many layers it takes is debated but it is safe to say that when there are three or more hidden layers it is getting "deeper." Another way that an ANN can become more complex is the direction in which information can move within the network. This is similar to how information moves forward and backwards in the human body. In an ANN this flow of information is called propagation, which was mentioned earlier. This complexity or flexibility allows ANN to be used for such things as classifying pictures to even driving cars.

Pros and Cons of Artificial Neural Networks

ANNs love large amounts of data and sometimes this can be hard to come by. There is a limit to how accurate other algorithms can be when you provide massive amounts of data. However, with ANN the more data there is an improvement in accuracy. Furthermore, with enough data an ANN will usually outperform any other algorithm.

However, due to the need for data, ANNs can be hard to train. In addition, the computational time can be excessive, which makes it hard to develop models quickly. Lastly, models that are too simple often will not converge yet it is hard to know if your model is too simple until you realize it is never going to work.

An entire book could be written on ANN. For our purpose, the rest of this chapter will look at a simple application of ANN in classification and regression.

ANN for Classification

We are going to use the `Wages` dataset. Our goal is to classify if someone is a member of a union or not based on the independent variables that are available in the dataset.

For classification and regression we will be using a multi-layer perceptron algorithm. This is a logistic regressor that includes a hidden layer in which the output is feed forward. By feed forward it means that the output never goes backwards in the model to a previous layer, which is possible with other types of ANNs.

Data Preparation

Here are the libraries we will be using.

```
from pydataset import data
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.neural_network import MLPClassifier
from sklearn import metrics
```

Next, we will load the data and take a look at it.

```
In [145]: df=data("Wages")
     ...: df.head()
     ...:
Out[145]:
   exp  wks bluecol  ind south smsa married   sex union  ed black    lwage
1    3   32      no    0   yes   no     yes  male    no   9    no  5.56068
2    4   43      no    0   yes   no     yes  male    no   9    no  5.72031
3    5   40      no    0   yes   no     yes  male    no   9    no  5.99645
4    6   39      no    0   yes   no     yes  male    no   9    no  5.99645
5    7   42      no    1   yes   no     yes  male    no   9    no  6.06146
```

There are several categorical variables. We need to convert these to dummy variables in order for the algorithm to work. We will use the `get_dummies` function to do this. Below is the code.

```
df=pd.get_dummies(df)
df.head()
```

The printout is too long but you can see it in your computer if you are following along.

Now, we will create our X and y datasets. For the X dataset, in addition to the continuous variables, we will only take the categorical variables that are labeled as yes. For the y dataset, we will only take the variable `union_yes`. We will then count the number union and non union examples in the dataset. The code is below.

```
In [150]: X=df[['exp','wks','ind','ed','lwage','south_yes',
     ...:        'smsa_yes','married_yes','sex_male','black_yes']]
     ...: y=df['union_yes']
     ...: y.value_counts()
     ...:
Out[150]:
0    2649
1    1516
Name: union_yes, dtype: int64
```

The final step is to divide our data into train and test sets.

```
X_train,X_test,y_train,y_test=train_test_split(X,
                                               y,test_size=.3,
                                               random_state=0)
```

We can now move to model development

Model Development

We need to create an instance of our classifier. We are going to set the following arguments for the algorithm.

- Hidden layer size-The size of the network

• Max iterations-Max number of attempts to reach convergence

• Activation-There are several types. We are using the relu. For more information see documentation on the internet

• Solver-There are several choices here. For us we are using the adam solver, which is stochastic gradient based solver

• Random state-Same as seed number allows for reproducible analysis

Below is the code.

```
clf=MLPClassifier(hidden_layer_sizes = (5, 2),
                  max_iter=300,activation = 'relu',
                  solver='adam',random_state=1)
```

We will now train our model and make the predictions.

```
clf.fit(X_train,y_train)
y_pred_train=clf.predict(X_train)
```

Lastly, we can check how our model.

```
In [161]: print(pd.crosstab(y_train,y_pred_train))
     ...: round(metrics.accuracy_score(y_train,y_pred_train),2)
     ...:
col_0         0    1
union_yes
0          1586  245
1           626  458
Out[161]: 0.7
```

The model has an accuracy of 70%. We can now see how this model will perform using the test data.

Model Testing

There is little to add in this section. We will predict with the test data and

check the metrics. Below is the code.

```
In [162]: y_pred_test=clf.predict(X_test)
     ...: print(pd.crosstab(y_test,y_pred_test))
     ...: round(metrics.accuracy_score(y_test,y_pred_test),2)
col_0          0    1
union_yes
0            704  114
1            263  169
Out[162]: 0.7
```

The performance is identical. Deciding if this is acceptable or not really depends on the goals of the project. There are many other things we could do to try and tweak the model but the purpose here is just demonstration. We will now turn our attention to regression.

ANN for Regression

We are going to use the Wages dataset for a second time but this time for regression. Our goal is to predict wages based on the other variables available in the dataset.

Data Preparation

The data preparation is mostly the same as in classification. The only difference is the dependent variable. Below are the libraries we will use.

```
import pandas as pd
from pydataset import data
from sklearn.model_selection import train_test_split
from sklearn.neural_network import MLPRegressor
import numpy as np
from sklearn.metrics import mean_squared_error
import matplotlib.pyplot as plt
import seaborn as sns
```

The only difference is we loaded the `MLPRegressor` library, a library

to calculate the mean squared error and some libraries for visualizing the data. The code below completes the data preparation process and was explained previously.

```
df=data("Wages")
df=pd.get_dummies(df)
X=df[['exp','wks','ind','ed','south_yes','smsa_yes',
      'married_yes','sex_male','black_yes','union_yes']]
y=df['lwage']
X_train,X_test,y_train,y_test=train_test_split(X,y,
                                               test_size=.3,
                                               random_state=0)
```

We can now move to model development.

Model Development

In the code below, we make an instance of our model, run it, and make the predictions. The settings for the arguments are the same as for the classification model.

```
mlp_reg=MLPRegressor(hidden_layer_sizes = (5, 2),
                     max_iter=300,activation = 'relu',
                     solver='adam',random_state=1)
mlp_reg.fit(X_train,y_train)
y_pred_train=mlp_reg.predict(X_train)
```

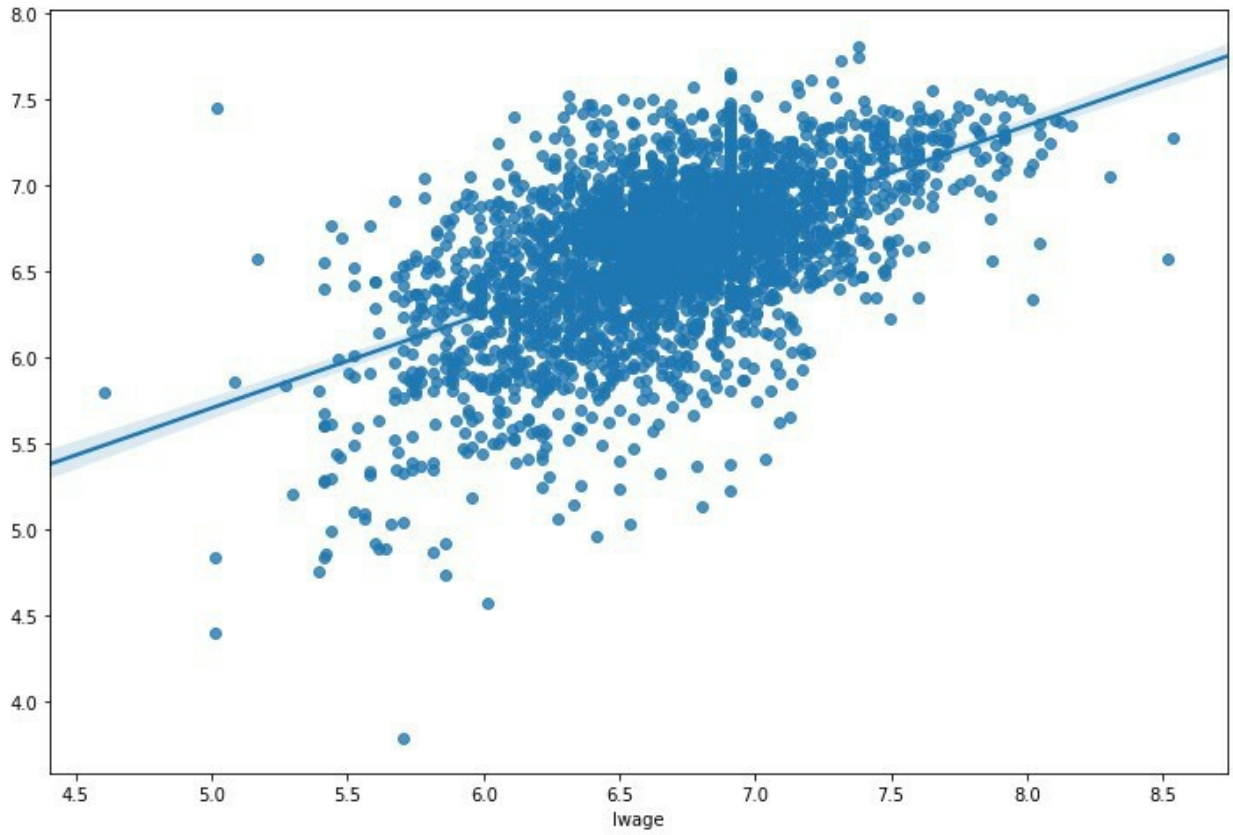
We will now see the performance of the model with the training data. The metrics we will use are the correlation between the actual and predicted values and the mean squared error. Below is the code and output for the correlation.

```
np.corrcoef(y_pred_train,y_train)
```

0.56

The correlation is moderate. Below is the code and scatterplot.

```
fig, ax = plt.subplots(figsize=(12,8))
sns.regplot(y_train, y_pred_train)
```



There appears to be some sort of a trend in the data. Below is the code and output for the mean squared error.

```
round(mean_squared_error(y_train, y_pred_train),5)
0.19249
```

We will now look at the performance of the model with the test data.

Model Testing

In the code below, we make the predictions with the test data and then calculate the correlation.

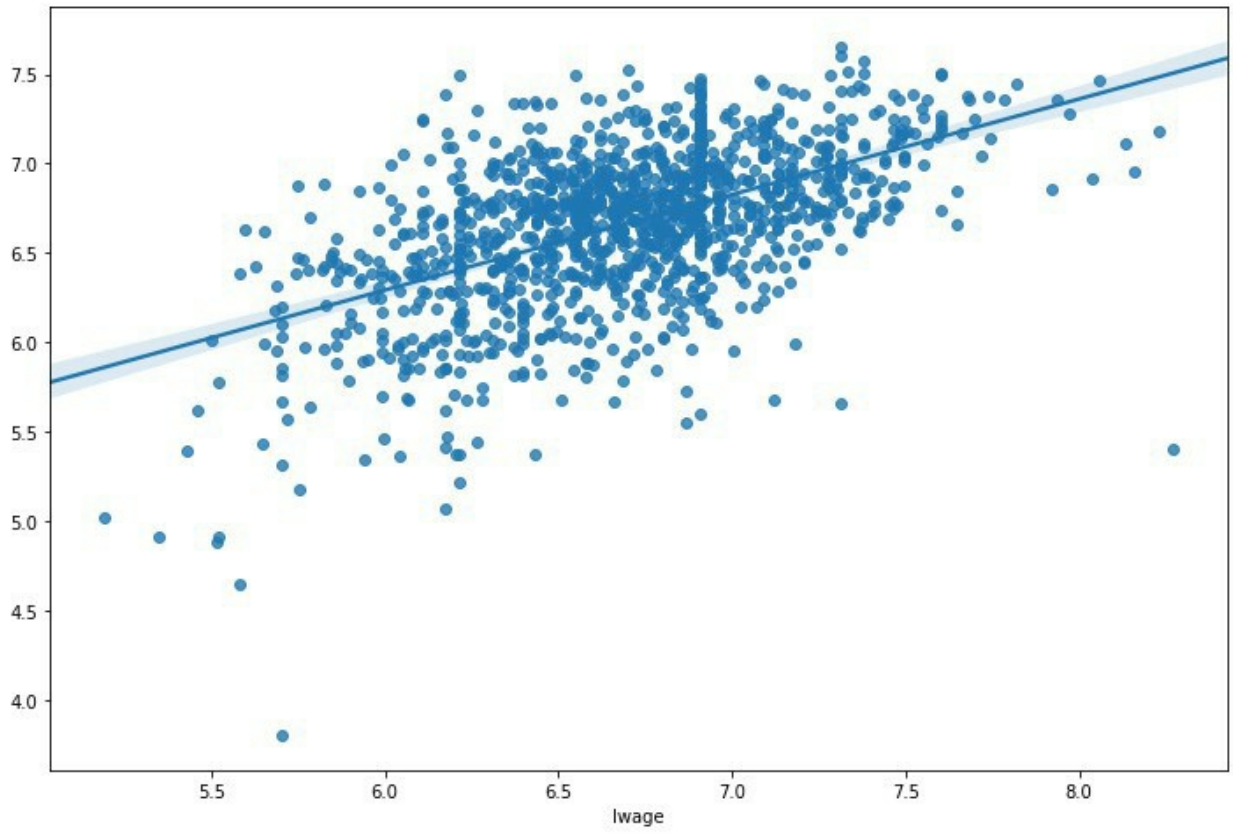
```
y_pred_test=mlp_reg.predict(X_test)
```

```
np.corrcoef(y_pred_train,y_train)
```

```
0.54
```

The correlation is almost exactly the same. Now we will make the scatterplot.

```
fig, ax = plt.subplots(figsize=(12,8))
sns.regplot(y_test, y_pred_test)
```



Finally, here is the MSE

```
round(mean_squared_error(y_test, y_pred_test),5)
0.18076
```

The MSE is also highly identical as well.

Conclusion

This chapter introduce the use of ANN. As with several other algorithms in this book, ANNs can be used for classification and regression. What was covered here was just an overview as ANNs have a lot of different types and ways to tune and adjust their performance.

Chapter 9: K-Means

In this chapter, you will learn the following...

• Explain the characteristics of Kmeans.

• Explain the pros and cons of Kmeans.

• Develop a model using Kmeans.

• How to interpret the results of Kmeans.

Introduction

K-means is an example of unsupervised learning. When learning is unsupervised it means that there is no dependent variable that we can use to determine how well are model is doing. In other words, there is no data that serves as the ground truth for the model to learn how to adjust the parameters. This means that it is left to the researcher to try and determine when the analysis is sufficient.

What k-means essentially does is it randomly throws out k number of centroids. The centroids are then moved using euclidean distance or some other metric until they are in the middle of a group of examples that are similar in nature based on the variables in the model. When the centroids stop moving the process is completed. The number of centroids or k to use is left to the researcher to determine. We will go through one example of how to do this.

K-means allows you to find patterns in the data that can be used to provide insights for action. Once the clusters are made. You can see what makes them different by checking descriptive statistics. This gives you a different view of your data compared to the other methods that we have discussed so far.

Since this is unsupervised learning, the process we will use will be different in this chapter. Below are the steps we will take.

1. Data preparation
2. Determine number of clusters
3. Develop model
4. Check descriptive statistics
5. Make a visual

Data Preparation

Our goal in this chapter is to use the `sat.act` dataset to make clusters based on the variables available in the dataset. Below is the code for the libraries used in this chapter.

```
from pydataset import data
from sklearn.cluster import KMeans
from scipy.spatial.distance import cdist
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
```

Next, we need to load our dataset `sat.act`, drop the NAs, and normalize the data. Normalization is necessary because the variables are measured in different units and the various scales can influence the outcome since we are use euclidean distance. Below is the code.

```
In [67]: df=data("sat.act")
    ...: df=df.dropna(how='any')
    ...: df.info()
    ...: df = (df - df.min()) / (df.max() - df.min())
    ...: df.head()
    ...:
<class 'pandas.core.frame.DataFrame'>
Int64Index: 687 entries, 29442 to 39985
Data columns (total 6 columns):
gender       687 non-null int64
education    687 non-null int64
age          687 non-null int64
ACT          687 non-null int64
SATV         687 non-null int64
SATQ         687 non-null float64
dtypes: float64(1), int64(5)
memory usage: 37.6 KB
Out[67]:
       gender  education       age       ACT      SATV      SATQ
29442     1.0        0.6  0.115385  0.636364  0.500000  0.500000
29457     1.0        0.6  0.192308  0.969697  0.666667  0.500000
29498     1.0        0.6  0.134615  0.545455  0.466667  0.450000
29503     0.0        0.8  0.269231  0.696970  0.583333  0.533333
29504     0.0        0.4  0.384615  0.848485  0.666667  0.583333
```

The data is now prepare. We will now turn our attention to determine how many clusters we need.

Determine the K

We now need to determine how many clusters are most optimal. The code below is hard to explain but it serves the purpose of determining the k.

```
distortions = []
K = range(1,10)
for k in K:
    kmeanModel = KMeans(n_clusters=k).fit(df)
    distortions.append(sum(np.min(
            cdist(df, kmeanModel.cluster_centers_,
                  'euclidean'), axis=1)) / df.shape[0])
```

Here is what happen.

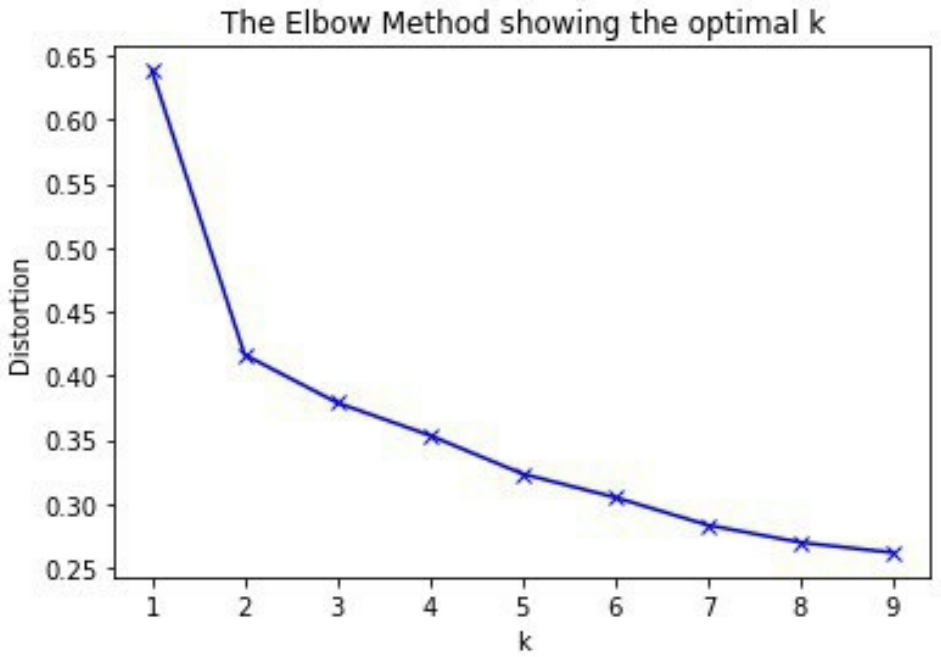
1. Made a list called `distortions`

2. Ran for loop with K being set from 1 to 10.

3. Inside the loop, we fit the k-means algorithm and calculated the sum of the euclidean distance of each cluster.

The code below provides a visual of the results of the k calculation.

```
plt.plot(K, distortions, 'bx-')
plt.xlabel('k')
plt.ylabel('Distortion')
plt.title('The Elbow Method showing the optimal k')
plt.show()
```



As you look at the plot, what you want to do is try and find the elbow and that tells you how many clusters to consider. It appears that there is a clear elbow at two clusters. However, for our purposes, we will set the k to 3. This will provide a more nuance interpretation.

Model Development

We are now going to create our model. We will use the Kmeans method. Inside this method, we have to set the number of clusters to three, set the initialization which will be 'k-means++', and set the random state to 0. All this is stored in an object called km. Next, we use the `.fit` method with the `df.values`. By adding values we convert our dataframe into an array temporarily. Then we use the `.predict()` method on our km object with the `df.values` inside in order to determine which cluster each sample belongs to. Lastly, we view the results using the `.head()` method. Below is the code and the results.

```
In [42]: km=KMeans(3,init='k-means++',random_state=0)
    ...: km.fit(df.values)
    ...: df['cluster']=km.predict(df.values)
    ...: df.head()
    ...:
Out[42]:
       gender  education       age       ACT      SATV      SATQ  cluster
29442     1.0        0.6  0.115385  0.636364  0.500000  0.500000        2
29457     1.0        0.6  0.192308  0.969697  0.666667  0.500000        2
29498     1.0        0.6  0.134615  0.545455  0.466667  0.450000        2
29503     0.0        0.8  0.269231  0.696970  0.583333  0.533333        1
29504     0.0        0.4  0.384615  0.848485  0.666667  0.583333        1
```

You can see that there is a new column called clusters. This is the result of the model development in that the computer has determine which cluster each sample belongs to.

The next step is to look at the descriptive statistics. The reason why there is no model testing in this chapter is there is no dependent variable that we have classification or continuous data for. Without this ground truth there is not much to test. This is why we are using other tools to understand the model.

Descriptive Statistics

The purpose of checking the descriptive statistics is that it allows us to see what are the differences among the clusters. This will provide us with insights into what is happening in each cluster. In order to do this we want to use the clusters we created using the scaled data. To do this we need to create another dataset with the original unscaled data and then add the cluster data to this. This will provide more meaning in terms of the characteristics of the clusters then in looking at scaled data. Below is the code and output for this process.

```
In [72]: df2=data("sat.act")
    ...: df2=df2.dropna(how='any')
    ...: df2['cluster']=df['cluster']
    ...:
    ...: round(df2.groupby(['cluster']).agg({'ACT':['mean'],'age':['mean'],
    ...:             'SATV':['mean'],'SATQ':['mean'],'education':['mean']}),2)
    ...:
Out[72]:
           ACT    age    SATV    SATQ education
          mean   mean    mean    mean      mean
cluster
0        29.34  32.53  618.17  600.02      4.41
1        28.82  25.92  615.36  635.87      3.00
2        27.69  20.12  604.94  592.94      2.39
```

Here is what we did.

1. The first two lines are just for importing the original dataset. We called the new dataset `df2` and drop any missing data.

2. In the third line, we map the cluster results from the df object to the `df2` object.

3. The next line of code we group the data by cluster and report the means for all of the continuous variables.

The output provides some insights. Cluster 0 has the oldest students who also have the most education and the highest test scores across most tests. Cluster 1 provides medium results while also being the second oldest and educated. Last, is cluster 2 which is much younger than the other clusters and has a

much poorer performance on all tests.

That was the results for the continuous variables. However, we can also check the results for the categorical variable of gender. This is done with a simple confusion matrix. The code is below.

```
In [74]: print(pd.crosstab(df2['cluster'],df2['gender']))
gender     1    2
cluster
0          0  191
1        245    0
2          0  251
```

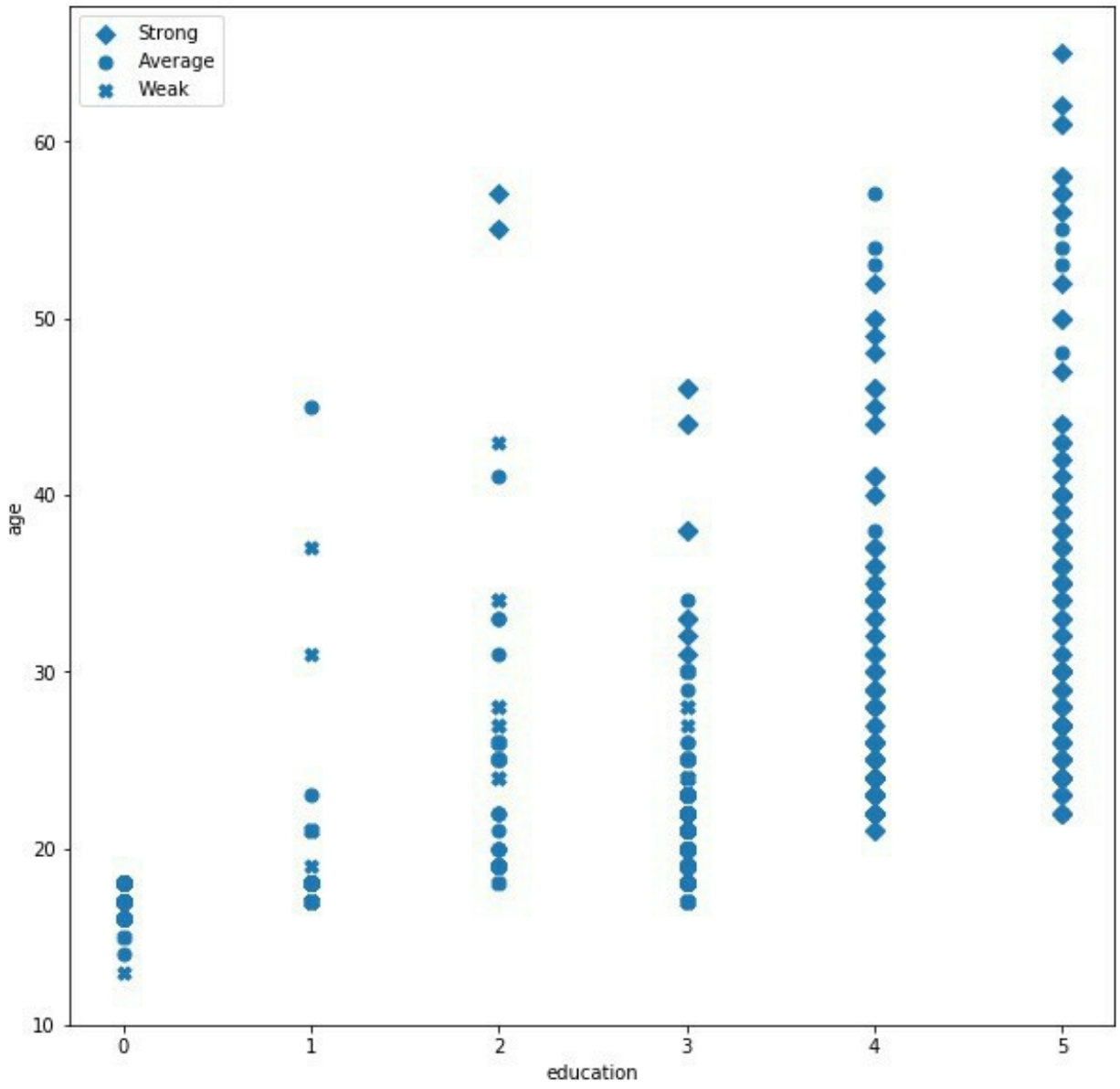
You can see there is a clear separation by gender. All the males are in the second cluster. While all the females are in clusters 0 and 2. This would explain the high quantitative SAT score of cluster 1. Men are often better at math then women and at least in this data this is confirmed. This provides us with an additional insight that cluster 0 is older, educated women. Cluster 1 is young males, and cluster 2 is very young women.

The next section will provide a visual of the clusters.

Visualization

Below is the code and visualization.

```
clust_map={0:'Strong',1:'Average',2:'Weak'}
df2['perf']=df2.cluster.map(clust_map)
d_color={'Weak':'X','Average':'o','Strong':'D'}
fig,ax=plt.subplots(figsize=(10, 10))
for clust in clust_map.values():
    marker=d_color[clust]
    df2[df2.perf==clust].plot(kind='scatter',
      x='education',y='age',label=clust,ax=ax,marker=marker,s=50)
```



Here is what happen.

1. In the first line, we create an interpretation of the clusters for the plot.

2. Line two creates the cluster map.

3. Line 3 sets the colors for each cluster.

4. The rest of the code is standard for make a visualization. In our case, we were plotting education and age with the categorical variable of our

three different clusters mapped as a third dimension.

It’s hard to see in the text but it is clear that as education and age increase the strong cluster becomes more common. The Average cluster or male cluster is spread all over the plot. This is common with mean as the spread for intellectual performance is much broader for men then for women.

Conclusion

K-means is a great tool for looking for ways to segment your data so that you can develop insights about your sample. If you are looking for patterns in your data and lack a clear sense of prediction this may be a strong and practical tool for you to consider.

Chapter 10: Assessing Classification Model Performance

In this chapter, you will learn the following...

- Interpret various metrics for classification model performance
- Use visuals to assess classification model performance
- Use cross-validation to assess the generalizability of a classification model

Introduction

In this chapter, we will look at how to assess the performance of a classification model such as a model that uses decision trees, random forest, or support vector machines.

The data we use in this chapter is the same data from the classification models we made in prior chapters. We will essentially reuse the code from the chapter on decisions trees and assess the model in a way that is much more thorough. Some of the ideas presented here are review but the purpose of having a separate chapter was to go deeper in how this can be approached after you were already familiar with basic ways of approaching this.

Data Preparation

The code below is the same as in the chapter on decision tree classification. It is reprinted here for convenience. Below are the libraries we will use.

```
import pandas as pd
from pydataset import data
import statsmodels.api as sm
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.metrics import (classification_report,
                             confusion_matrix,
                             accuracy_score,cohen_kappa_score)
import seaborn as sns
from matplotlib.colors import ListedColormap
from sklearn import preprocessing
import numpy as np
from sklearn.metrics import roc_curve, auc
from sklearn.model_selection import (KFold, cross_val_score)
from sklearn import tree
```

Here we drop any NA, set setup the different datasets and recode the categorical variable.

```
df = data('cancer')
df=df.dropna()
X=df[['time','age',"sex","ph.ecog",'ph.karno',
      'pat.karno','meal.cal','wt.loss']]
df['status']=df.status.replace(1,'censored')
df['status']=df.status.replace(2,'dead')
y=df['status']
```

Lastly, we create our train and test sets.

```
x_train, x_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
                                                    random_state=0)
```

We will now turn our attention to model development.

Model Development

Below is the code for the decision tree and the prediction results with the training data.

```
In [50]: clf=tree.DecisionTreeClassifier(max_depth=3)
    ...: clf=clf.fit(x_train,y_train)
    ...: y_pred_train=clf.predict(x_train)
    ...:

In [51]: print (pd.crosstab(y_pred_train,y_train))
status    censored  dead
row_0
censored         4     0
dead            28    84

In [52]: round(accuracy_score(y_train,y_pred_train),2)
Out[52]: 0.76

In [53]: print(classification_report(y_train,y_pred_train))
              precision    recall  f1-score   support

    censored       1.00      0.12      0.22        32
        dead       0.75      1.00      0.86        84

   micro avg       0.76      0.76      0.76       116
   macro avg       0.88      0.56      0.54       116
weighted avg       0.82      0.76      0.68       116
```

The last line with the `classification_report` function is new and prints out several metrics that we will explain in a moment. All of these metrics are based on the confusion metrics, which is directly above. We will now examine the confusion matrix much more carefully along with the metrics that are derived from it.

Model Assessment

In this section, we will assess the model. We are going to explore the confusion matrix and all the metrics that can be derived from it as well as the ROC curve, which is a useful visualization of classification model quality.

Confusion Matrix

We will begin by looking at a confusion matrix. The code below is a more visually friendly version of what you have already seen.

```
from sklearn.externals.six import StringIO
from IPython.display import Image
from sklearn.tree import export_graphviz
import pydotplus
dot_data = StringIO()
export_graphviz(clf, out_file=dot_data,
                filled=False, rounded=True,feature_names=list(x_train.columns.values),
                class_names=list(y_train),special_characters=True)
graph = pydotplus.graph_from_dot_data(dot_data.getvalue())
Image(graph.create_png())
cm = confusion_matrix(y_pred_train,y_train)
ax= plt.subplots(figsize=(10,10))
sns.set(font_scale=3.4)
with sns.axes_style('white'):
    sns.heatmap(cm,
                cbar=False,
                square=True,
                annot=True,
                fmt='g',
                cmap=ListedColormap(['gray']),
                linewidths=2.5)
```

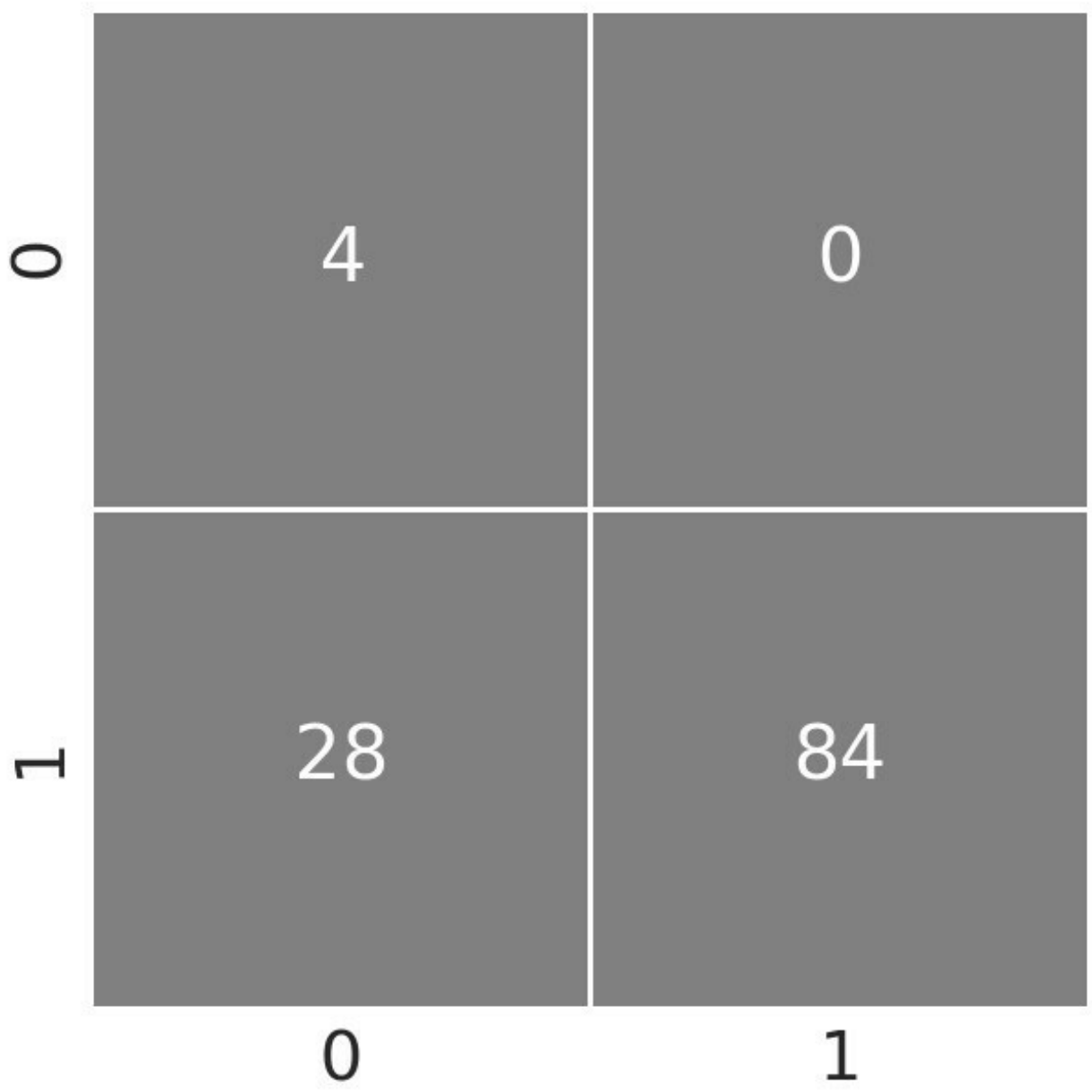


Figure 10.1: Confusion Matrix

You've seen the output in the table above but we have not really explained it explicitly. A confusion matrix is a table that is used to organize the predictions made during an analysis of classification data.

The example above is a two-class confusion matrix. This matrix compares the actual class of an example with the predicted class of the model. Below is a breakdown of what is happening in each region of the confusion matrix.

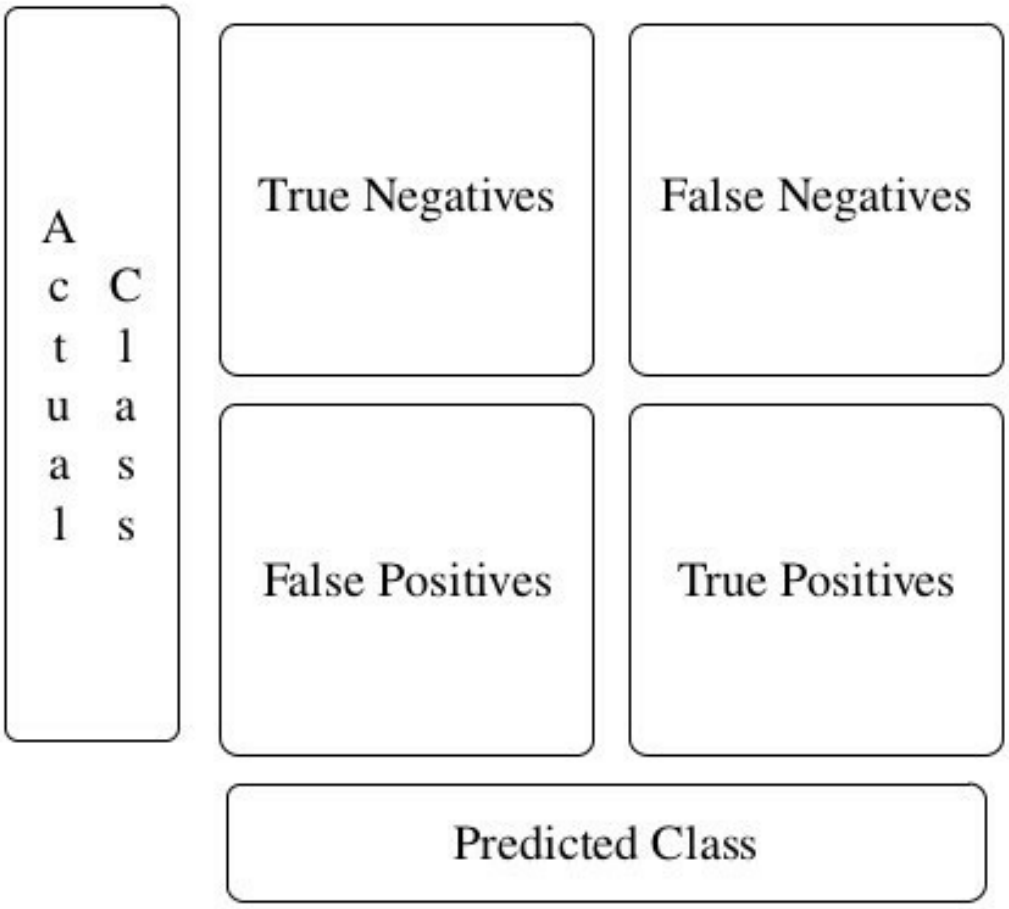


Figure 10.2: Abstract Confusion Matrix

If we take what we know from our output and compare it to a blank confusion matrix this is what we learn.

1. In the top left we have true negatives. These are examples that were correctly classified as not having the characteristic we were looking for. In our example, there were 4 people who were classified as not dead and predicted as not dead.

2. In the top right we have false negatives. These are examples that were incorrectly classified as not having the characteristic we were looking for but actually did have the characteristic. In our example, there were 0 people who were classified as being dead but were predicted as not being dead. In other words, there were no false negatives

3. In the bottom left we have our false positives. These are examples that incorrectly classified as having the characteristics we are looking for but actually do not possess it. In our example, 28 people who were not dead were predicted as being dead.

4. In the bottom right corner we have our true positives. These are examples that were correctly classified as having the characteristic we are looking for. In our example, 84 people were classified as dead who were predicted as being dead.

Confusion Matrix Metrics

There are a lot of different metrics that can be calculated using the information from the confusion matrix. We will look at the following...

- Accuracy
- Error
- Precision
- Recall/Sensitivity
- F-measure
- Specificity
- Kappa

Accuracy

Accuracy is the total number of items correctly classified divided by the total number of items. Below is the expression.

$$\text{accuracy} = \frac{TP+TN}{TP+TN+FP+FN}$$

TP = true positive, TN = true negative, FP = false positive, FN = false

negative

Below is the code for calculating the accuracy in Python.

```
round(accuracy_score(y_train, y_pred),2)
```

```
Out[83]:
```

```
0.76
```

This is review from prior chapter but as you can see the accuracy in our model was 76%. Accuracy can range in value from 0-1 with one representing 100% accuracy. Normally, you don't want perfect accuracy as this is an indication of over-fitting and your model will probably not do well with other data.

Error

is 1 minus the accuracy. It indicates how many examples were misclassified. Below is the expression.

$$\text{error} = 1 - \frac{TP+TN}{TP+TN+FP+FN}$$

Below is the code for calculating the error in Python.

```
round(1 - accuracy_score(y_train, y_pred),2)
```

```
Out[84]:
```

```
0.24
```

The error in our model was 24%.

For precision, recall/sensitivity, and f-measure we can use the `classification_report` function as this was done earlier in the chapter to get these metrics all at the same time. Below is the output for reference.

	precision	recall	f1-score	support
censored	1.00	0.12	0.22	32
dead	0.75	1.00	0.86	84
micro avg	0.76	0.76	0.76	116
macro avg	0.88	0.56	0.54	116
weighted avg	0.82	0.76	0.68	116

There are a lot of numbers here. The columns contain the names of the categories followed by different types of averages (more on this in a minute). To the right are the three main metrics we will discuss right now, which are precision, recall, and the f1-score. The last column just indicates how many samples were classified in either category. The first row explains how well the model did classifying people as ’censored.’ The second row indicates how well the model did classifying people as ’dead’. Below the first two rows you have the same metrics but broken down into micro, macro, and weighted averages. Here is a break down of what each of these averages mean.

Micro

Calculate metrics for the entire sample by counting the total true positives/negatives, false negatives and false positives.

Macro

Calculate metrics for each category, and find their unweighted mean. This does not take into account any imbalance among the categories.

Weighted

Calculate metrics for each category, and finds their average weighted by support (the number of true instances for each category). This accounts for category imbalance.

For our purposes, we will focus on the micro average as we go through all of the metrics.

Precision

is the proportion of examples that are really positive. The formula is as follows

$$\text{precision} = \frac{TP}{TP+FP}$$

The more precise a model is the more trustworthy it is. In other words, high precision indicates that the results are relevant. For our model, it was .76.

Sensitivity/Recall

is the proportion of true positives that were correctly classified. The formula is as follows

$$\text{sensitivity} = \frac{TP}{TP+FN}$$

This may sound confusing but high sensitivity is useful for assessing a negative result. In other words, if I am trying to test people for a disease and my model has a high sensitivity. This means that the model is useful telling me a person does not have a disease or the results are negative. The term 'recall' is the term used for sensitivity in a different context such as for analyzing the strength of a web search engine. For our model, the recall/sensitivity was .76.

F-measure

uses recall and precision to develop another way to assess a model. The formula is below

$$\text{F-measure} = \frac{2*TP}{2*TP+FP+FN}$$

The f-measure can range from 0 - 1 and is useful for comparing several

potential models using one convenient number. For our model, the f-measure is 0.76.

You are probably wondering why all the metrics values are the same in our example. This is due in part to the fact that we have 0 false negatives. This is why there is so much redundancy in the values. Essentially we are calculating all of these metrics without including the influence of the false negatives.

In addition to the metrics above there are also several metrics that the python function does not calculate, and these are listed below.

Specificity

measures the proportion of negative examples that were correctly classified. The formula is below.

$$\text{specificity} = \frac{TN}{TN+FP}$$

Returning to the disease example, a high specificity is a good measure for determining if someone has a disease if he or she test positive for it. The specificity for our model was 0.13, which is not the best value.

Kappa

This statistic is a measurement of accuracy of a model while taking into account chance. The closer the value is to 1 the better. Below is the code for calculating the kappa.

```
round(cohen_kappa_score(y_train,y_pred_train),3)

Out[87]:

0.171
```

This kappa of our model is rather poor and indicates that our model is not that good at prediction when controlling for guessing.

Different metrics are more important in different situations. Sometimes

sensitivity is important while at other times specificity is more important. What effects this is the context and whether you are more concern with true positives or true negatives. The purposes of these different metrics is to provide you with different insights into judging models you will make.

ROC Curve

The receiver operating characteristic curve (ROC curve) is a tool used in statistical research to assess the trade-off of detecting true positives and true negatives. The history of this tool goes all the way back to World War II when radar engineers were trying to distinguish between true and false alarms when examining radar data of enemy plane movements. You can imagine the cost or danger of either sending planes out when there are no enemies or not sending planes out when there are enemies.

Today, the ROC Curve is used in machine learning not for determining the presence of enemy planes but rather to assess the ability of a model to detect true positives and negatives. The ROC curve is plot that shows a half "t" shape when it is strong. Below in figure 10.3 is an example of what a strong ROC curve looks like.

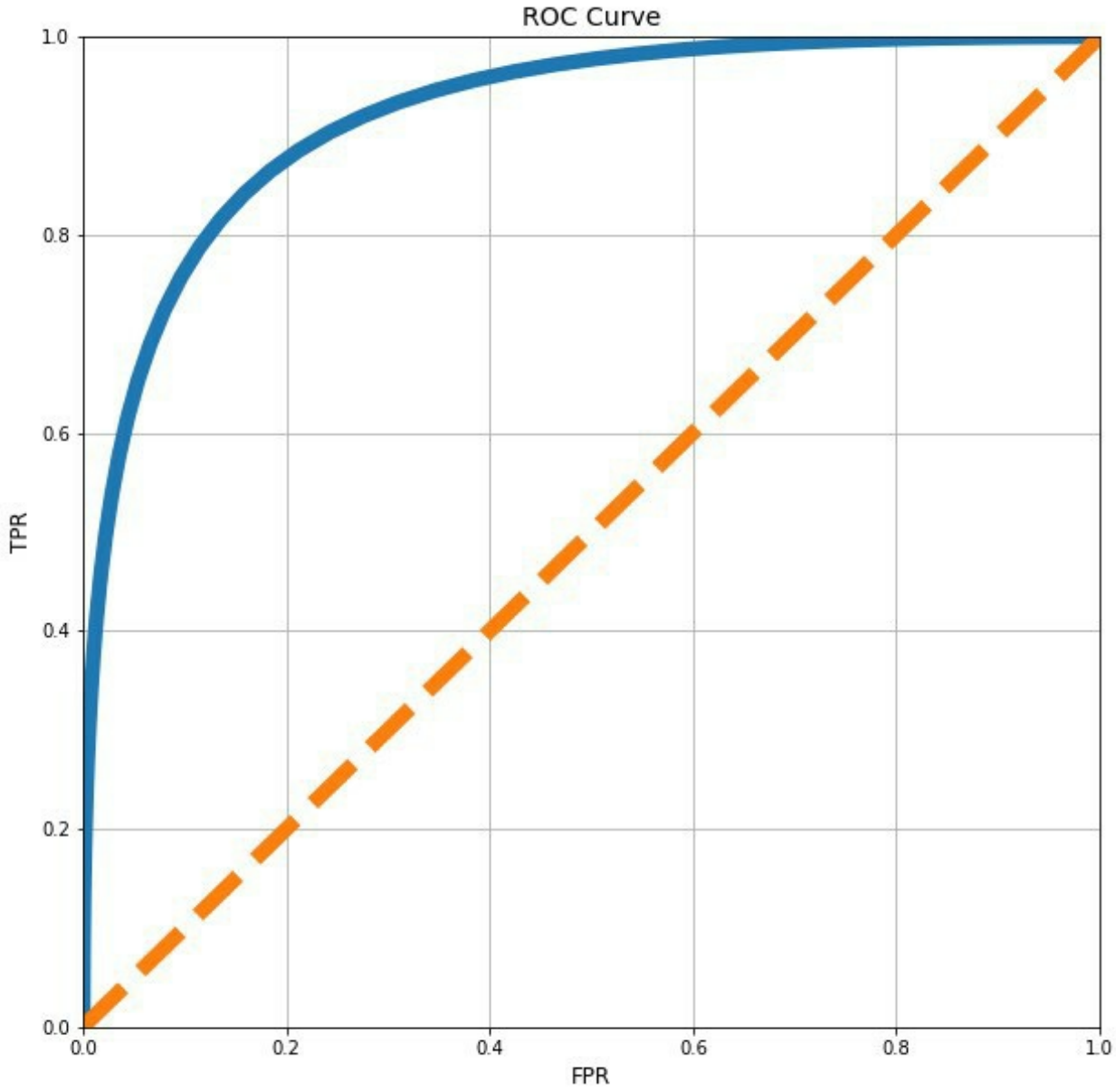


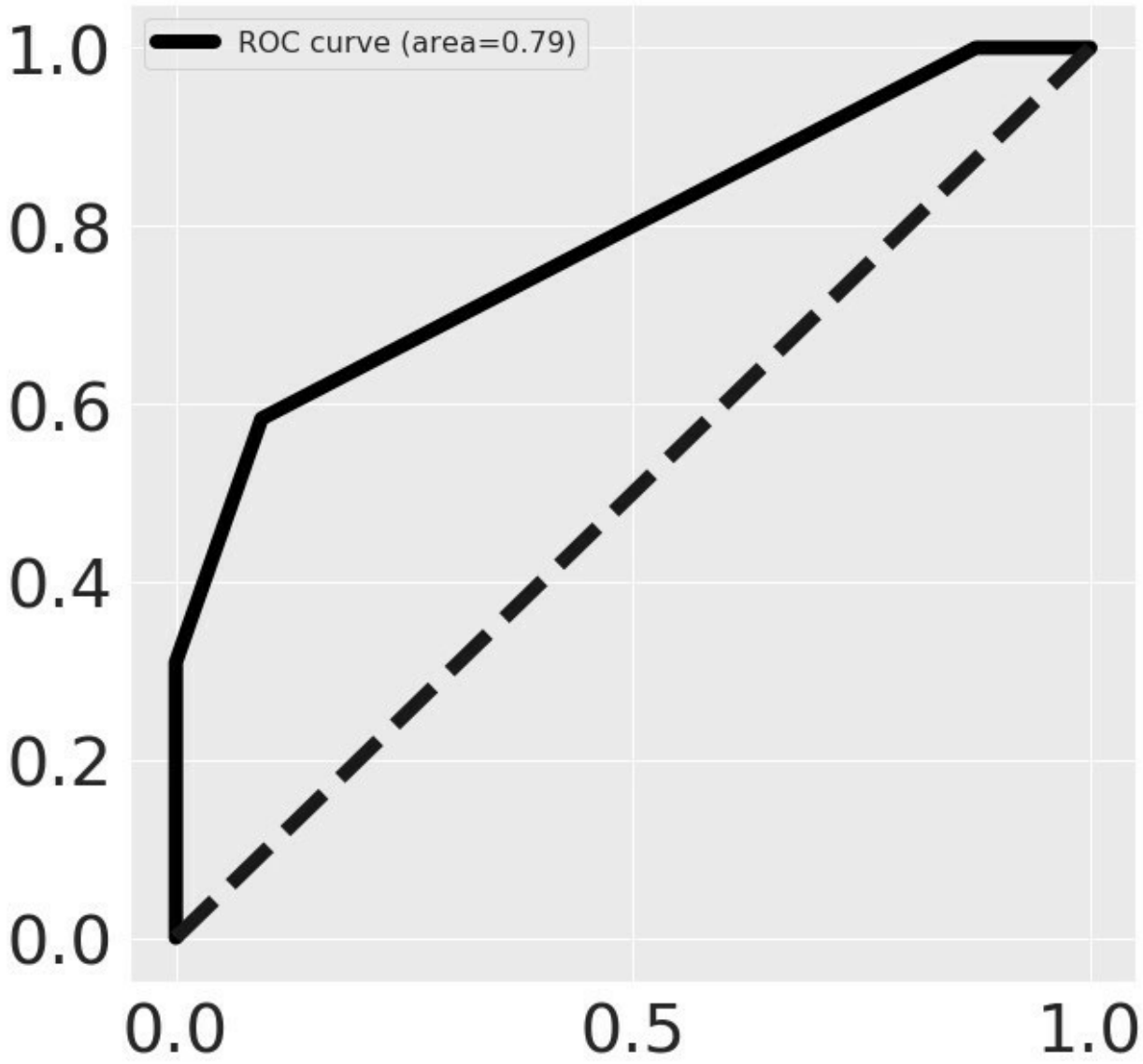
Figure 10.3: Strong vs Weak ROC Curve

The solid line is the fictitious "good" model. The dash line is what a "poor" model would look like. You can clearly see the strong half "t" shape of the model. The more diagonal the line the worst the model is.

Below is the code for calculating the ROC curve and a graph of the results of our model. One difference you will see is that we predict the probabilities of each class with the `.predict_proba()` method rather than simply predict the class with the `.predict()` method. This is because the functions used to

create the ROC curve needs this information. As for the rest of the code, we will not explain it due to its complexity.

```
y_pred_train_prob=clf.predict_proba(x_train)
y_train=y_train.replace('dead',1)
y_train=y_train.replace('censored',0)
fpr, tpr, thresholds = roc_curve(y_train,
                                 y_pred_train_prob[:,1],
                                 pos_label=1)
roc_auc = auc(fpr, tpr)
fig, ax = plt.subplots(1,1, figsize=(10,10))
plt.clf()
plt.plot(fpr,tpr,
         label="ROC curve (area=%0.2f)" % roc_auc,
         linewidth=8,color='black')
plt.plot([0,1],[0,1],'k--',linewidth=8)
plt.legend(loc="upper left",prop={'size': 16})
```



You can see that our model line possess at least somewhat that strong half "t" shape. This means that the ROC curve of our model is not that strong, which means that the model is having some success with finding true positives and true negatives. This is further supported by the ROC curve value of 0.79. The ROC curve can have a value between 0-1. A score of 0.79 is consider acceptable for this metric.

Cross-Validation

The final tool that we will use is cross-validation. This tool divides our training set into k number of folds (normally 5 or 10). One fold is excluded and the data is train on the remaining folds. The fold that was excluded is then used as a test set. This is done k times and the results are averaged.

This process helps in seeing how stable the results are and if they would generalize to other datasets. In order to do this in Python, first we need to make an instance of the cross validation and this is shown below.

```
crossvalidation=KFold(n_splits=10,
                      shuffle=True,random_state=1)
```

Next, we need to calculate the different folds. To do this we will use `cross_value_score` function. The first argument is the model (we named this clf at the beginning of the chapter). This is followed by our training sets, our scoring criteria is accuracy, our cv will be cross validation, and the number of jobs will be one. The code is below.

```
scores=cross_val_score(clf,
                       x_train,y_train,
                       scoring='accuracy',
                       cv=crossvalidation,n_jobs=1)
```

The next piece of code has the actual results. We are using the functions below to how many folds we made (10) the mean, and standard deviation. Below is the code.

```
In [96]: print(len(scores),round(np.mean(np.abs(scores)),2),
    ...:       round(np.std(scores),2))
10 0.73 0.11
```

We used accuracy as the criteria for the cross-validation. The final results show that we had about 73% accuracy with a standard deviation of about 11%. This is not much different from our model that did not use cross-

validation. The standard deviation is rather large but this may be because our sample size is rather small.

You may be wondering why we never used the test data. This is because the test data should only be used once. Since we were trying to assess the quality of our model it was not appropriate to use the test data. Once we have decided what model to use it is only then that we use the test data.

Conclusion

This chapter explained how to assess a classification model using various metrics that are derived from a confusion matrix. These tools are meant to guide and inform your decision-making as you try to make sense of the data and models that you have developed.

Chapter 11: Assessing Regression Model Performance

In this chapter, you will learn the following...

- Interpret various metrics for regression model performance
- Use visuals to assess regression model performance
- Use cross-validation to assess generalizability of a regression model

Introduction

In this chapter we will look at various ways to assess the performance of a regression model or a model that makes numeric predictions. The tools used here are not exhaustive. However, they will help you to get started when it is time to assess a model.

Many of these tools have been used in prior chapters. However, to make things as clear as possible the various tools that were applied before are now all in one place for practical and pedagogical reasons.

To demonstrate the tools in this chapter we are going to use the numeric prediction decision tree that we developed in chapter 4. Our goal in that chapter was to predict the age of cancer patients using the independent variables. Below is a list of the various ways we will assess the performance of a regression model in this chapter.

- Several metrics
- Correlation of predicted with actual values
- Cross-validation

Data Preparation

Below are the libraries we will use in this chapter.

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.metrics import (mean_squared_error,r2_score)
from pydataset import data
from sklearn.model_selection import KFold
from sklearn.model_selection import cross_val_score
import seaborn as sns
from sklearn import tree
```

The rest of the preparation is loading the dataset, separating the dependent from the independent variables, and creating the train and test sets. The code for all of this is below.

```
df = data('cancer')
df=df.dropna()
X=df[['time',"sex","ph.ecog",'ph.karno','pat.karno',
      'meal.cal','wt.loss']]
y=df['age']

x_train, x_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
                                                    random_state=0)
```

Model Development

Below is the code for the model development followed by the output of the model. This is still a review of prior content.

```
reg=tree.DecisionTreeRegressor(max_depth=3)
reg=reg.fit(x_train,y_train)
```

In order to continue to assess the model we need to predict the y variable. This is done with the following code.

```
y_pred_train=reg.predict(x_train)
```

Model Assessment

Let's begin by looking at descriptive statistics. The goal of this analysis is to compare the mean, standard deviation, and quartile statistics of the actual values with the predicted values. The general rule is that they should be of similar value. Below is the code and the output.

```
In [49]: y_train.describe()
Out[49]:
count    116.000000
mean      63.370690
std        9.405392
min       39.000000
25%       57.750000
50%       64.500000
75%       70.000000
max       82.000000
Name: age, dtype: float64

In [50]: pd.DataFrame(data=y_pred_train.flatten()).describe()
Out[50]:
                0
count  116.000000
mean    63.370690
std      5.111134
min     39.000000
25%     58.548387
50%     64.040816
75%     64.959184
max     72.888889
```

```
fig, ax = plt.subplots(figsize=(12,8))
sns.regplot(y_train, y_pred_train)
```

You can see that the `.describe()` method is used each time. The `t_pred_train` object required additional coding because we needed to convert a numpy array to a panda dataframe.

The results are mostly similar. The means are the same. There are some differences in the standard deviations and the 75% percentiles. However, the actual and predicted values have similar measures.

We will now move to something that you are already familiar with which is a visualization. We will make a scatterplot of the actual and predicted values. After the scatterplot we will look at the actual correlation between these two statistics. Below is the code and visualization for this.

```
fig, ax = plt.subplots(figsize=(12,8))
sns.regplot(y_train, y_pred_train)
```

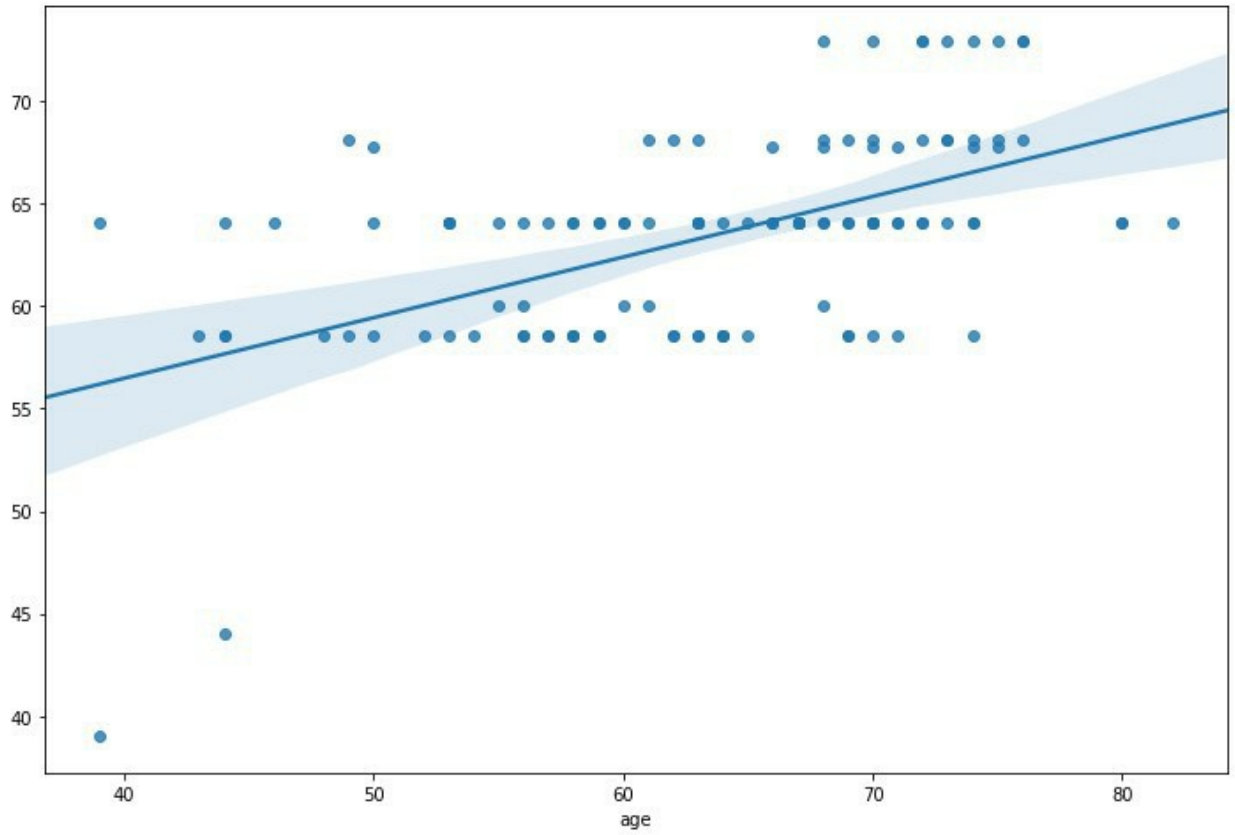


Figure 11.1: Correlation of Actual & Predicted Values

This looks promising. Below is the actual correlation between the actual and predicted values.

```
np.corrcoef(y_pred_train,y_train)
```

```
0.54
```

The correlation is moderate which indicates that our model may be good at predicting future values. The last two metric that we will calculate are the mean squared error (MSE)and the r^2. The MSE is a comparison metric which means it is only useful in comparing models and has little value by itself. The better model normally has the lower MSE. The r^2 has a range of 0-1 and indicates the strength of a model. Below is the code and output.

```
In [55]: round(mean_squared_error(y_train, y_pred_train),3)
Out[55]: 61.8

In [56]: round(r2_score(y_train,y_pred_train),3)
Out[56]: 0.295
```

The MSE means nothing by it's self. As for the r^2, a value of 0.29 is not that exciting in most context. We would need to revise the model if we want to consider this metric.

Cross-Validation

Cross-validation is tool that was discussed in the last chapter. We are

going to use this tool with the MSE and the r^2.

In this section, we will do cross-validation twice. Once with the MSE being the criteria and a second time with r^2 being the criteria. First, we will examine the MSE. The code below is the same as the previous chapter. With the minor change being the argument `scoring` to `neg_mean_squared_error`.

```
In [62]: crossvalidation=KFold(n_splits=10,shuffle=True,random_state=1)
    ...: scores=cross_val_score(reg,
    ...:                        x_train,y_train,
    ...:                        scoring='neg_mean_squared_error',
    ...:                        cv=crossvalidation,n_jobs=1)
    ...: print(len(scores),round(np.mean(np.abs(scores)),2),
    ...:       round(np.std(scores),2))
    ...:
10 88.6 30.34
```

The MSE is higher than in the training data. However, the real problem may be in the standard deviation. A value of 30 is high given the value of the MSE. This might be because the sample size is rather small (116). A high standard deviation means that the values are spread far from the mean. This high spread means that we are probably struggling to explain the variance and this is confirmed in the r^2 we calculated earlier. Below is the code and output for the cross-validation of r^2.

```
In [64]: scores=cross_val_score(reg,
    ...:                        x_train,y_train,
    ...:                        scoring='r2',
    ...:                        cv=crossvalidation,n_jobs=1)
    ...: print(len(scores),round(np.mean(np.abs(scores)),3),
    ...:       round(np.std(scores),3))
10 0.207 0.198
```

The average or expected r^2 is 0.2 with a standard deviation of 0.19, which is really high as with what we found with the MSE.

Conclusion

Assessing model performance is a critical part of data science and machine learning. The reason for this is to determine if the model created is just meaningless numbers or a useful tool.

Chapter 12: Model Improvement Strategies

In this chapter, you will learn the following...

- Explain tools for assessing model performance for classification.
- Explain tools for assessing model performance for numeric prediction.

Introduction

Now that we have some idea of how well our model performs there are several ways to improve a model. Some of these strategies are more practical than others. Below is a partial list of ways to improve a model.

- Add variables
- Remove variables
- Create variables
- Change algorithms
- Collect more data
- Adjust hyperparameters

Each of these ideas have their strengths and weaknesses. Adding variables might work if you have other data that was not included in the original model. Removing variables can also improve performance but you are loosing information/dimensionality when this is done. Creating variables or feature generation is somewhat complex yet creative but requires domain expertise to make features that make sense conceptually for the context. Changing algorithms can also work but brings new problems of having to make adjustments to accommodate the characteristics of different algorithms. Lastly, collecting more data can only improve so much performance and may not be possible. However, hyperparameter tuning does not involve any of these problems and allows us to assess our model with existing resources.

A hyperparameter in machine learning is a value that is set before the algorithm begins the learning process. Each model or algorithm has a different hyperparameter(s) that can be tuned. For example, K nearest neighbor has hyperparameters for the number of neighbors, the distance, weights, etc. For support vector machines the hyperparameter that you can tune is "C" which stands for cost function, which is the penalty for misclassification or numeric prediction error. In addition other hyperparameters that could be adjusted include the kernel, degree, gamma,

etc.

Our goal here will be to demonstrate how tune a classification model and then how to tune a numeric prediction model. We will tune our KNN classification model from chapter 5. Then we will tune our support vector machine numeric prediction model from chapter 7.

Classification Hyperparameter Tuning

In the code below, we load our libraries, create the dataset, normalize the data, recode the dependent variable, create the independent and dependent datasets, and split the data. There is nothing new in this code except for the `GridSearchCV` library, which will be explained in a moment.

```
from pydataset import data
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn import metrics
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import GridSearchCV

df=data("turnout")
df=df.drop('race',1)

df = (df - df.min()) / (df.max() - df.min())

X=df[['age','educate','income']]
df['vote']=df.vote.replace(1,'yes')
df['vote']=df.vote.replace(0,'no')
y=df['vote']

X_train,X_test,y_train,y_test=train_test_split(X,y,test_size=.3,
                                               random_state=0)
```

Original Model

Below is the code for the original model. This will serve as our baseline to determine if the tuning of the hyperparameters improves things.

```
In [82]: clf=KNeighborsClassifier(7)
    ...: clf.fit(X_train,y_train)
    ...: y_pred_train=clf.predict(X_train)
    ...: print(pd.crosstab(y_train,y_pred_train))
    ...: round(metrics.accuracy_score(y_train,y_pred_train),2)
col_0   no  yes
vote
no     136  218
yes     80  966
Out[82]: 0.79
```

It appears our current goal is to try and beat an accuracy of 79%.

Hyperparameter Tuning

There are several steps we need to take in order to tune the hyperparameters.

1. Create a grid with different hyperparameter settings.

2. Let Python search the grid.

3. Determine the best hyperparameter settings.

There really is no magic way to set the hyperparameters. Instead, what is commonly done is you select various ranges or options and run all of them and see which combination is the best. This means that you need to know what hyperparameters are available for whatever algorithm you are using. For KNN classification, these are the hyperparameters we want to tune.

- Number of neighbors

- Weights

- Distance metrics

- Power for one of the distance metrics

Now we will make a grid. The grid provides a way to assess multiply hyperparameter combinations. Below is the code for creating a grid.

```
grid={'n_neighbors':range(1,13),'weights':['uniform','distance'],
      'metric':['manhattan','minkowski'],'p':[1,2]}
```

Essentially we are telling python the following

- Set the number of neighbors anywhere from 1 to 13
- Set the weights to uniform or distance
- Set the distance metric to manhattan or minkowski
- Set power for one of the distance metrics to 1 or 2

Python will run every possible combination of hyperparameters developing a model for each one. In case you are curious, below is how many models we will make.

```
np.prod([len(grid[element]) for element in grid])

Out[85]: 96
```

We can now allow python to search the grid. This is done by first make an instance using the GridSearchCV function. We have to include in this function our original model object clf, along with the grid we created, the scoring metric, number of jobs, refit, and the cross-validation k. After completing this we will use the `.fit()` method with the training data. The code is below.

```
search=GridSearchCV(estimator=clf,
                    param_grid=grid,scoring='accuracy',
                    n_jobs=1,refit=True,cv=10)
search.fit(X_train,y_train)
```

Finally, we can determine the best parameter combination with the code below.

```
In [94]: print(search.best_params_)
    ...: print(round(search.best_score_,2))
    ...:
{'metric': 'minkowski', 'n_neighbors': 12, 'p': 2, 'weights': 'distance'}
0.74
```

The results tell us the following,

- Set the number of neighbors to 12
- Set the weights to distance
- Set the distance metric to minkowski
- Set power for one of the distance metrics to 2

The cross-validate results indicate that we can expect an accuracy of 74% if we use this combination of hyperparameters.

Modified Model

The cross-validation of the model already tells us what to expect if we wanted to use our model with new data. However, for the sake of completeness, we will modify our original model with the tuned hyperparameters and see what the results are. Below is the code and results.

```
In [100]: clf2=KNeighborsClassifier(n_neighbors=12,
     ...:                            weights='distance',
     ...:                            metric='minkowski',p=2)
     ...: clf2.fit(X_train,y_train)
     ...: y_pred_train=clf2.predict(X_train)
     ...: print(pd.crosstab(y_train,y_pred_train))
     ...: round(metrics.accuracy_score(y_train,y_pred_train),3)
     ...:
col_0   no   yes
vote
no     354     0
yes      4  1042
Out[100]: 0.997
```

You can see how amazing the results are compare to the original model. Now the accuracy is 99.7% compared to the original accuracy of 79%. However, this might be too good as such a high accuracy indicates overfitting. One reason the accuracy is so high is because we are assessing the accuracy on data that we trained the model on. The cross-validated results of 74% show us that we would not have such a high accuracy on new data.

Numeric Prediction Hyperparameter Tuning

We will now tune a numeric prediction model. We will use the support vector machine algorithm from an earlier chapter. Below is the initial code for the libraries and data preparation. This is all review.

```
In [112]: import numpy as np
     ...: from pydataset import data
     ...: from sklearn import svm
     ...: from sklearn.metrics import mean_squared_error
     ...: from sklearn import model_selection
     ...: import pandas as pd
     ...: from sklearn.model_selection import GridSearchCV

In [113]: import numpy as np
     ...: from pydataset import data
     ...: from sklearn import svm
     ...: from sklearn.metrics import mean_squared_error
     ...: from sklearn import model_selection
     ...: import pandas as pd
     ...: from sklearn.model_selection import GridSearchCV
     ...:
     ...: df=pd.DataFrame(data('Workinghours'))
     ...: df=df.dropna()
     ...:
     ...: df=df.drop(['occupation'],1)
     ...: df['all_kids']=df['child5']+df['child13']+df['child17']
     ...:
     ...: df = (df - df.min()) / (df.max() - df.min())
     ...:
     ...: X=df[['hours','age','income','all_kids',
     ...:       'nonwhite','owned','unemp']]
     ...: y=df['education']
     ...:
     ...: x_train,x_test,y_train,y_test=model_selection.train_test_split(X,y,test_size=.
3,random_state=1)
     ...:
```

Original Model

Below is the code for the original model. This will serve as our baseline to determine if the tuning of the hyperparameters improves performance.

```
h1=svm.SVR()
h1.fit(x_train,y_train)
y_pred_train=h1.predict(x_train)
```

```
In [120]: round(mean_squared_error(y_train, y_pred_train),5)
Out[120]: 0.01589

In [121]: np.corrcoef(y_train,y_pred_train)
Out[121]:
array([[1.        , 0.46991934],
       [0.46991934, 1.        ]])
```

Our primary metric is the MSE. However, we also can see the correlation between the actual and predicted values. Are goal is to reduce the MSE and strengthen the correlation but the actual and predicted values.

Hyperparameter Tuning

SVM has a different set of hyperparameters. We are not going to set all of them but below are the ones we will address.

- Value for the cost function
- Type of kernel
- Degree

Below is the code for creating a grid.

```
search_grid=[{'C':[1,3,10],'kernel':['linear','rbf'],'degree':[1,2,3]}]
scorer='neg_mean_squared_error'
```

Essentially we are telling Python the following

- Set the value for the cost function to 1, 3 or 10
- Set the type of kernel to linear or rbf
- Set the degree to 1, 2 or 3

Python will run every possible combination of hyperparameters developing a model for each one. The code is below.

```
search_func=model_selection.GridSearchCV(estimator=h1,
                                         param_grid=search_grid,
                                         scoring=scorer,n_jobs=-1,
                                         refit=True,cv=10)
search_func.fit(x_train,y_train)
```

Finally, we can determine the best parameter combination with the code below.

```
In [153]: print(search_func.best_params_)
     ...: print(round(search_func.best_score_,5))
     ...:
{'C': 10, 'degree': 1, 'kernel': 'rbf'}
-0.01598
```

The results tell us the following,

- Set the value for the cost function to 10
- Set the type of kernel to rbf
- Set the degree to 1 (this is the default value)

The cross-validate results indicate that we can expect a MSE of 0.01598 (take the absolute value) if we use this combination of hyperparameters. This is similar to the original model's MSE of 0.01589.

Modified Model

We will now take our new hyperparameter settings and use them. Below is the code and results.

```
h2=svm.SVR(C=10, degree=1,kernel='rbf')
h2.fit(x_train,y_train)
y_pred_train2=h2.predict(x_train)
```

```
: np.corrcoef(y_train,y_pred_train2)
:
1.         , 0.49362148],
0.49362148, 1.         ]])

: round(mean_squared_error(y_train, y_pred_train2),5)
: 0.01538
```

There is some improvement in the model. The MSE has gone down from 0.01589 to 0.01538. Whether this is good or bad depends on the dependent variable. If this was a reduction in millions of dollars it might make a big deal as such, this difference could be important depending on the context. The correlation is also stronger but not as much of an improvement as we would hope for.

Conclusion

Determining the value of hyperparameters is a matter of trial and error. Every situation is different and unique and it takes experiences to determine what to do. The examples in this chapter showed you how to tune hyperparameters for classification and numeric prediction using different algorithms. The only way to get better is to use these tools yourself.

Chapter 13: Combining Algorithms

In this chapter, you will learn the following...

• How to combine several algorithms for classification.

• How to assess the performance of a combined algorithm.

Introduction

In this chapter, we tie together everything that has been covered in this book. We will combine several different algorithms (random forest, support vector machines, KNN) to predict the gender of people in the `DoctorAUS` dataset as we did in the chapter 4. Through combining algorithms you are able to maximize the strengths while minimizing the weakness of the different algorithms you use.

The technique that we will use to combine algorithms is called stacking. Stacking involves making predictions with several models and the results of these models are passed to a higher model that uses the individual model predictions to make a final prediction.

A note of caution, explaining every step is too complicated at this point. Therefore, it is left to you to follow along and try and understand what is happening with the code and the results. The following steps will be taken to accomplish our goal

1. Data preparation

2. Individual model development

3. Ensemble model development

4. Hyperparameter tuning of the ensemble model

5. Ensemble model testing

Data Preparation

Below are the libraries that are used in this chapter.

```
from pydataset import data
from sklearn.model_selection import train_test_split
from sklearn.model_selection import cross_val_score
from sklearn import svm
from sklearn.ensemble import RandomForestClassifier
from sklearn.neighbors import KNeighborsClassifier
from mlxtend.classifier import EnsembleVoteClassifier
from sklearn.metrics import accuracy_score
from sklearn.metrics import classification_report
from sklearn.model_selection import GridSearchCV
import pandas as pd
```

In the code below, we do the following...

- Load the dataset
- Create our independent dataset and drop unwanted variables
- Replace the numbers in the dependent variable with text
- Create our dependent dataset
- Create our train and test sets

```
df = pd.DataFrame(data('DoctorAUS'))
X=df.drop(['sex','insurance', 'chcond'], axis = 1)
df['sex']=df.sex.replace(1,'female')
df['sex']=df.sex.replace(0,'male')
y=df['sex']
x_train, x_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
                                                    random_state=0)
```

We will now move to model development

Individual Model Development

We will now create instances of our models. We will not fit the models yet as this is not necessary at this moment. We will then use loops to assess the accuracy and the area under the curve. Below is the code for making the models.

```
rfclf=RandomForestClassifier()
svmclf=svm.SVC(kernel='sigmoid')
knnclf=KNeighborsClassifier(n_neighbors=5,p=2,metric='minkowski')
```

Below is the for loop and the results for the accuracy. Note that these results and most of the results in this chapter are cross-validated

```
In [124]: clf_labels=['Random Forest','SVM','KNN']
     ...: for clf, label in zip ([rfclf,svmclf,knnclf],clf_labels):
     ...:     scores=cross_val_score(estimator=clf,X=x_train,
     ...:                            y=y_train,cv=10,scoring='accuracy')
     ...:     print("accuracy: %0.2f (+/- %0.2f) [%s]" % (scores.mean(),
     ...:                             scores.std(),label))
     ...:
     ...:
accuracy: 0.65 (+/- 0.03) [Random Forest]
accuracy: 0.59 (+/- 0.02) [SVM]
accuracy: 0.62 (+/- 0.02) [KNN]
```

You can see that the random forest model is the best performing one. None of the models are that strong but the svm model is struggling the most. Below is the for loop and results for the auc.

```
In [125]: for clf, label in zip ([rfclf,svmclf,knnclf],clf_labels):
     ...:     scores=cross_val_score(estimator=clf,X=x_train,
     ...:                            y=y_train,cv=10,scoring='roc_auc')
     ...:     print("roc auc: %0.2f (+/- %0.2f) [%s]" % (scores.mean(),
     ...:                            scores.std(),label))
     ...:
     ...:
roc auc: 0.69 (+/- 0.02) [Random Forest]
roc auc: 0.59 (+/- 0.02) [SVM]
roc auc: 0.65 (+/- 0.02) [KNN]
```

You can see the auc values. None are all that impressive. Individually, the

models are not that strong. We will now move to assessing the performance of the combined model

Ensemble Model Development

We do the following in this section.

1. Fit each of our individual models and create the ensemble model
2. Check the accuracy of the ensemble model
3. Check the auc of the ensemble model

First, we will fit the individual models and create the ensemble model. You can weigh the models differently so we will give more weight to the random forest model because it is the strongest.

```
rfclf.fit(x_train,y_train)
svmclf.fit(x_train,y_train)
knnclf.fit(x_train,y_train)
mv_clf= EnsembleVoteClassifier(clfs=[rfclf,svmclf,knnclf],
                               weights=[1.5,1,1])
```

Second, we check the accuracy of the combined model. Note that the individual models' accuracy are reprinted again as well.

```
In [127]: labels=['Random Forest','SVM','KNN','combine']
     ...: for clf, label in zip ([rfclf,svmclf,knnclf,mv_clf],labels):
     ...:     scores=cross_val_score(estimator=clf,X=x_train,
     ...:                            y=y_train,cv=10,scoring='accuracy')
     ...:     print("accuracy: %0.2f (+/- %0.2f) [%s]" % (scores.mean(),
     ...:                             scores.std(),label))
     ...:
     ...:
accuracy: 0.65 (+/- 0.02) [Random Forest]
accuracy: 0.59 (+/- 0.02) [SVM]
accuracy: 0.62 (+/- 0.02) [KNN]
accuracy: 0.64 (+/- 0.02) [combine]
```

The combined model is worst than the random forest model. This is not good but we are learning here. Below is the results for the auc.

```
In [197]: for clf, label in zip ([rfclf,svmclf,knnclf,mv_clf],labels):
     ...:     scores=cross_val_score(estimator=clf,X=x_train,
     ...:                            y=y_train,cv=10,scoring='roc_auc')
     ...:     print("roc auc: %0.2f (+/- %0.2f) [%s]" % (scores.mean(),
     ...:                            scores.std(),label))
     ...:
     ...:
roc auc: 0.69 (+/- 0.02) [Random Forest]
roc auc: 0.59 (+/- 0.02) [SVM]
roc auc: 0.65 (+/- 0.02) [KNN]
roc auc: 0.69 (+/- 0.02) [combine]
```

The auc value is also mundane. We will now turn our attention to hyperparameter tuning.

Hyperparameter Tuning

What we will do in this section is tune the hyperparameters for each model. To keep things simple, we will only tune one hyperparameter for each model. This step can take a long time if you try and tune many hyperparameters of many models. For the hyperparameters we are tuning max depth for randomforest, the cost function for svm, and the number of neighbors for KNN. In the code below we setup our grid and fit the model.

```
params={'randomforestclassifier__max_depth':[2,5],
        'svc__C':[0.001,1,10],
        'kneighborsclassifier__n_neighbors':[5,9,12]}
grid=GridSearchCV(estimator=mv_clf,param_grid=params,
                  cv=10,scoring='accuracy')
grid.fit(x_train,y_train)
```

Below are the results for the best hyperparameter settings and the accuracy to expect.

```
In [202]: grid.best_params_
Out[202]:
{'kneighborsclassifier__n_neighbors': 12,
 'randomforestclassifier__max_depth': 5,
 'svc__C': 1}

In [203]: round(grid.best_score_,3)
Out[203]: 0.678
```

If we set the number of neighbors for KNN to 12, the max depth of random forest to 5, and the cost function of svm to 1. We should expect an accuracy of almost 68%. Let’s move to the model testing with these hyperparameter settings.

Ensemble Model Testing

In this final section, we will use are tuned hyperparamters to see how are ensemble model performs with the train and test sets. Below is the code with the tune models and the fit of the ensemble model.

```
rfclf=RandomForestClassifier(max_depth=5)
svmclf=svm.SVC(kernel='sigmoid',C=1)
knnclf=KNeighborsClassifier(n_neighbors=12,p=2,metric='minkowski')
mv_clf= EnsembleVoteClassifier(clfs=[rfclf,svmclf,knnclf],
                               weights=[1.5,1,1])
mv_clf.fit(x_train,y_train)
```

Here is the results after making the predictions on the training data.

```
In [210]: y_pred=mv_clf.predict(x_train)

In [211]: print(round(accuracy_score(y_train,y_pred),3))
0.703

In [212]: print(pd.crosstab(y_train,y_pred))
col_0   female  male
sex
female    1416   486
male       592  1139

In [213]: print(classification_report(y_train,y_pred))
              precision    recall  f1-score   support

      female       0.71      0.74      0.72      1902
        male       0.70      0.66      0.68      1731

    accuracy                           0.70      3633
   macro avg       0.70      0.70      0.70      3633
weighted avg       0.70      0.70      0.70      3633
```

```
In [217]: y_pred_test=mv_clf.predict(x_test)

In [218]: print(round(accuracy_score(y_test,y_pred_test),3))
0.681

In [219]: print(pd.crosstab(y_test,y_pred_test))
col_0   female  male
sex
female     564   236
male       260   497

In [220]: print(classification_report(y_test,y_pred_test))
              precision    recall  f1-score   support

      female       0.68      0.70      0.69       800
        male       0.68      0.66      0.67       757

    accuracy                           0.68      1557
   macro avg       0.68      0.68      0.68      1557
weighted avg       0.68      0.68      0.68      1557
```

The accuracy is 70%. The confusion matrix is left for you to interpret.

Below are the results for the test data.

About the same, the sad things is that after all this work we didn't really improve the performance of our model. The random forest model

got similar results with no where near the effort. This is just an example of how complexity is not a cure all for better performance.

Conclusion

This chapter pull together all the ideas of this book. We experienced several different algorithms, model assessing, model improvement, cross-validation, hyperparameters, and even the combining of algorithms to boost performance. However, even with all this knowledge the model did not improve as much as we were hoping it would. The final message is simple is almost always better.

Python is a highly popular language used in programming that has ca large influence in the field of data science. In this text, Darrin Thomas presents various ways to use Python to for model development in the context machine learning. Being able to comprehend the use of Python for data science is a valuable skill for anyone who must deal with data on a regular basis.

ISBN-13:9781071450215